Realms: A Foundation for Spatial Data Types in Database Systems[1]

Ralf Hartmut Güting
Markus Schneider

Praktische Informatik IV, FernUniversität Hagen
Postfach 940, D-5800 Hagen, Germany
gueting@fernuni-hagen.de, schneide@fernuni-hagen.de

Abstract: Spatial data types or algebras for database systems should (i) be fully general (which means, closed under set operations, hence e.g. a region value can be a set of polygons with holes), (ii) have formally defined semantics, (iii) be defined in terms of finite representations available in computers, (iv) offer facilities to enforce geometric consistency of related spatial objects, and (v) be independent of a particular DBMS data model, but cooperate with any. We offer such a definition in two papers. The central idea, introduced in this (first) paper, is to use *realms* as geometric domains underlying spatial data types. A realm as a general database concept is a finite, dynamic, user-defined structure underlying one or more system data types. A geometric realm defined here is a planar graph over a finite resolution grid. Problems of numerical robustness and topological correctness are solved below and within the realm layer so that spatial algebras defined above a realm enjoy very nice algebraic properties. Realms also interact with a DBMS to enforce geometric consistency on object creation or update.

Keywords: Spatial data types, algebra, realm, finite resolution, numerical robustness, topological correctness, geometric consistency.

[1] This work was supported by the DFG (Deutsche Forschungsgemeinschaft) under grant Gu 293/1-1.

1 Introduction

We consider a *spatial database system* to be a full-fledged DBMS with additional capabilities for the representation and manipulation of geometric data. As such, it provides the database technology needed to support applications such as *geographic information systems*. The standard DBMS view for the organization of spatial information is the following: A database consists of several classes of objects. A *spatial object* is just an object with an associated value ("attribute") of a *spatial data type*, such as, for example, POINT, LINE, or REGION. This is true regardless of whether the DBMS uses a relational, complex object, object-oriented or some other data model. Hence the definition and implementation of spatial data types is probably the most fundamental issue in the development of spatial database systems.

Although spatial data types (SDTs) are used routinely in the description of spatial query languages (e.g. [LiN87, JoC88, SvH91]), have been implemented in some prototype systems (e.g. [RoFS88, OrM88, Gü89]), and some formal definitions have been given [Gü88a, ScV89, GaNT91], there is still no completely satisfactory solution available according to the following criteria:

- *Generality.* The geometric objects used as SDT values should be as general as possible. For example, a region value should be able to represent a collection of disjoint areas each of which may have holes. More precisely, this means that the domains of data types POINT, LINE, and REGION must be closed under union, intersection, and difference of their underlying point sets. This allows for the definition of powerful data type operations with nice closure properties.
- *Rigorous definition*. The semantics of SDTs, that is, the possible values for the types and the functions associated with the operations, must be defined formally to avoid ambiguities for the user and the implementor.
- *Finite resolution*. The formal definitions *must take into account the finite representations available in computers*. This has so far been neglected in definitions of SDTs. It is left to the programmer to close this gap between theory and practice which leads rather inevitably not only to numerical but also topological errors.
- *Treatment of geometric consistency.* Distinct spatial objects may be related through geometric consistency constraints (e.g. adjacent regions have a common boundary). The definition of SDTs must offer facilities to enforce such consistency.
- *General object model interface*. Spatial data types as such are rather useless; they need to be integrated into a DBMS data model and query language. However, a definition of SDTs should be valid regardless of a particular DBMS data model and therefore not depend on it.[2] Instead, the SDT definition should be based on an abstract interface to the DBMS data model which we call the *object model interface*.

The purpose of this paper (together with a companion paper) is to develop a formal definition of spatial data types fulfilling these criteria. The central idea is to introduce into the DBMS the concept of a *realm*. A realm is in general a finite, user defined structure that is used as a basis for one or more system data types. Realms are somewhat similar to enumeration types in programming languages. A

[2] This also holds for the implementation level: A spatial type extension package (STEP) should be able to cooperate with any extensible DBMS offering a suitable interface regardless of its data model.

realm used as a basis for spatial data types is essentially a finite set of points and *non-intersecting* line segments. Intuitively, it describes the complete underlying geometry of an application. All points, lines and regions associated with objects (from now on called *spatial attribute values*) can be defined in terms of points and line segments present in the realm. In fact, in a database spatial attribute values are then never created directly but only by selecting some realm objects. They are never updated directly. Instead, updates are performed on the realm and from there propagated to the dependent attribute values.

Hence, all attribute values occurring in a database are *realm-based*. Furthermore, the algebraic operations for the spatial data types are defined to construct only geometric objects that are realm-based as well. So the spatial algebra is closed with respect to a given realm. This means in particular that no two values of spatial data types occurring in geometric computation have "proper" intersections of line segments. Instead, two initially intersecting segments have already been split at the intersection point when they were entered into the realm. One could say that any two intersecting SDT values (say, lines or regions) "have become acquainted" already when they were entered into the realm. This is a crucial property for the correct and efficient implementation of geometric operations.

Realm objects - points and segments - are defined not in abstract Euclidean space but in terms of finite representations. All geometric primitives and realm operations (e.g. updates) are defined in error-free integer arithmetic. For mapping an application's set of intersecting line segments into a realm's set of non-intersecting segments the concept of redrawing and finite resolution geometry from [GrY86] is used. Although intersection points computed with finite resolution in general move away from their exact Euclidean position, this concept ensures that the unavoidable distortion of geometry (that is, the numerical error) remains bounded and very small and that essentially[3] no topological errors occur. This means that a programmer has a precise specification that directly lends itself to a correct implementation. It also means that the spatial algebra obeys algebraic laws precisely in theory as well as in practice. Furthermore, it is rather obvious that realms also solve the geometric consistency problem.

Most closely related to this work are the formal definitions of spatial data types (or algebras) given by Güting [Gü88a, Gü88b], Scholl and Voisard [ScV89, Vo92], and Gargano *et al.* [GaNT91]. All of these proposals do not fulfill most of the criteria given above. In [Gü88a, Gü88b] data types for points, lines, and regions are available but too restricted, e.g. a region is a single simple polygon (without holes). In [ScV89] general regions are defined; in Voisard's thesis [Vo92] this has been extended to general types for points and lines. However, the definitions are unnecessarily complex. In [GaNT91] there is only a single type for all kinds of geometric objects; a value is essentially a set of sets of pixels. We feel this is not sufficient, since many interesting spatial operations cannot be expressed. As mentioned, all of these proposals give formal definitions. However, those of Güting and of Scholl and Voisard are not based on finite resolution; hence the numeric correctness problems are not addressed. Gargano *et al.* base their definitions in principle on a finite underlying set (of pixels). But this is not practical since these finite representations are far too large to be efficiently manageable. The geometric consistency problem is not solved in any of these proposals; there is some weak support in [Gü88a] through an AREA data type, but it is not sufficient. Finally, all three proposals have connected their spatial types to a fixed data model – Güting and Gargano *et al.* to the relational model

[3] See the discussion in Sections 2 and 8.

and Scholl and Voisard to a complex object algebra [AbB88]. Only Scholl and Voisard emphasize a clean interface between the spatial algebra and the general object model. We shall extend their work by offering an abstract interface not dependent on any particular data model.

The topological data model based on simplicial complexes suggested by Egenhofer *et al.* [EgFJ89] has a similar purpose as our concept of realms. Essentially they offer an irregular triangular network partition of the plane as a geometric domain over which spatial objects could be defined. However, the connections are missing to the underlying finite arithmetic as well as to spatial data types based on this model. Also, in our view a triangular partition contains too much information; it is sufficient to keep those points and segments in a geometric domain that are needed for spatial attribute values. Finally, their model is an abstract one whereas we show realms within a database context.

Our description and formal development of *realm-based spatial data types* is given in two papers. In this paper the lower layers, namely numerically robust geometric primitives, realms and their operations and the structure of values of the spatial data types are defined. In [GüS92] the object model interface and the spatial data types and operations, that is, the spatial algebra, are described. In the following section we first provide an informal overview of the complete concept.

2 Overview: Realm-Based Spatial Data Types

A *realm* is a set of points and non-intersecting line segments over a discrete domain, that is, a grid, as shown in Figure 1.

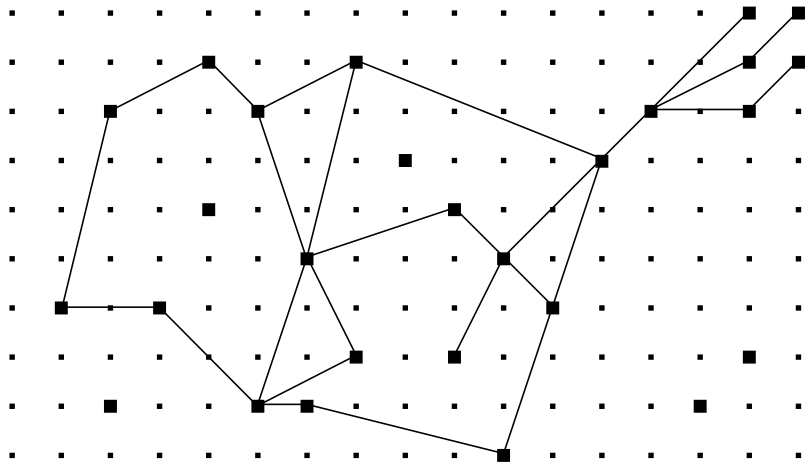


Figure 1

Values of spatial data types can be composed from the objects present in a realm. Figure 2 shows some values definable over the realm of Figure 1. Our realm-based spatial data types are called POINTS, LINES, and REGIONS, hence A and B represent REGIONS values, C is a LINES value, and D a POINTS value. The precise structure of these values is not yet relevant here. One can imagine A and B to belong to two adjacent countries, C to represent a river, and D a city.

The underlying grid of a realm arises simply from the fact that numbers have a finite representation in computer memory. In practice, these representations will be of fixed length and correspond to INTEGER or REAL data types available in programming languages. Of course, the resolution will be much finer than could be shown in Figure 1.

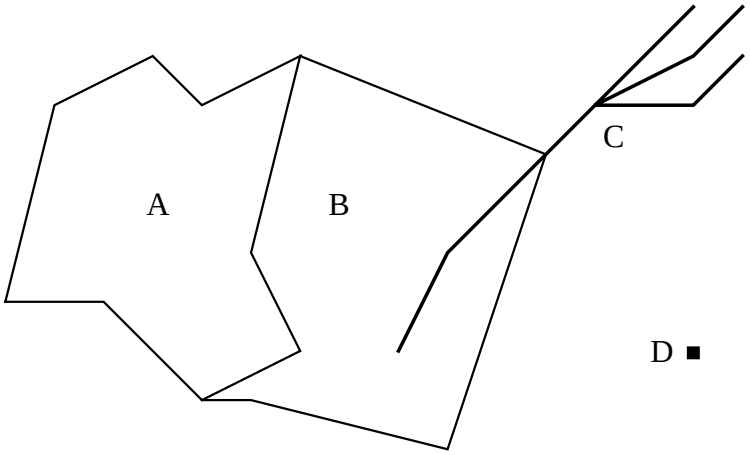


Figure 2

The concept of realm as a basis of spatial data types serves the following purposes:

- It enforces *geometric consistency* of related spatial objects. For example, the common part of the borders of countries A and B is exactly the same for both objects.
- It guarantees nice *closure properties* for the computation with spatial data types above the realm. For example, the intersection of region B with line C (the part of river C lying within country B) is also a realm-based LINES value.
- It shields geometric computation in query processing from numeric correctness and robustness problems. This is because such problems arise essentially from the computation of intersection points of line segments which normally do not lie on the grid. With realm-based SDTs, there are *never any new intersection points computed* in query processing. Instead, the numeric problems are treated *below* the realm level, namely, whenever updates are made to a realm.
- Additionally, a data structure representing a realm can be used as an index into the database. Our implementation concept assumes that each point and segment in a realm has an associated list of logical pointers to the spatial attribute values defined over it in the database.

Let us now focus on the treatment of numeric correctness problems below and within the realm level. This is necessary because geometric data coming from the application are not intersection-free, as required for a realm. Application data can at the lowest level of abstraction be viewed as a set of points and *intersecting* line segments. These need to be transformed into a realm. As mentioned before, the fundamental problem is that intersection points usually do not lie on the grid.

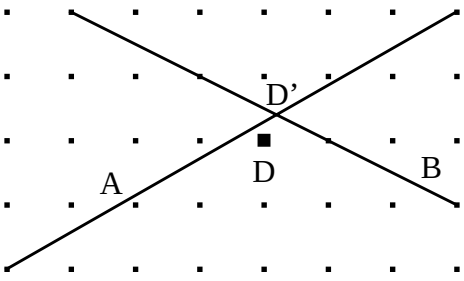


Figure 3

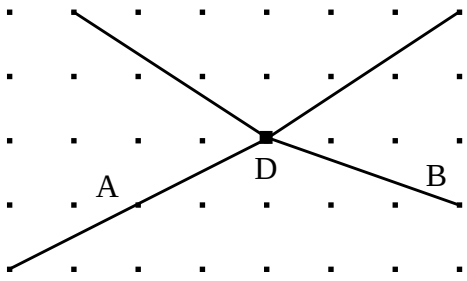


Figure 4

In Figure 3, the intersection point D' of line segments A and B will be moved to the closest grid point D. This leads, for example, to the following topological errors: (1) A test whether D lies *on* A or B

fails. (2) A test whether D lies properly within some area defined below A and B will incorrectly yield *true*. (3) If there is another segment C between the true intersection point and D, D will be reported to lie on the wrong side of C. The basic idea to avoid these errors is to slightly change segments A and B by transforming them into chains of segments going through D, as shown in Figure 4. However, this does not suffice, since it allows a segment to drift (through a series of intersections) by an arbitrary distance from its original position. For example, a further intersection of A with some segment C (Figure 5) is resolved as shown in Figure 6, where intersection point E has already a considerable distance from the true intersection point of A and C. Note in particular, that segment A has in Figure 6 been moved to the other side of a grid point (indicated by the arrow) which may later be reported to lie on the wrong side of A.

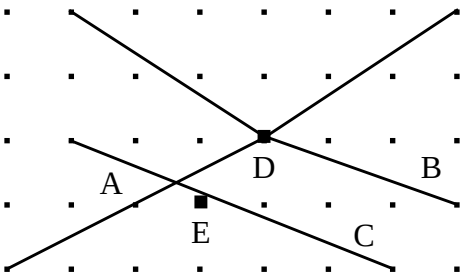


Figure 5

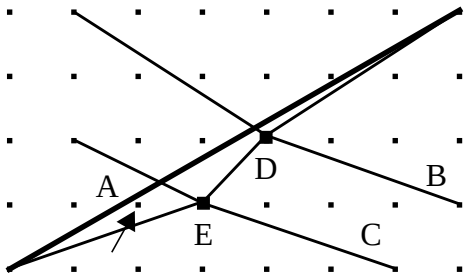


Figure 6

A refined solution was proposed by Greene and Yao [GrY86]. The idea is to define for a segment *s* an *envelope E*(*s*) roughly as the collection of grid points that are immediately above, below, or on *s*. An intersection of *s* with some other segment may lead to a requirement that *s* should pass through some point *P* on its envelope (the grid point closest to the true intersection point). This requirement is then fulfilled by *redrawing s* by some polygonal line *within the envelope* rather than by simply connecting *P* with the start and end points of *s*. Figure 7 shows a segment *s* (drawn fat) together with the grid points of its envelope. Slightly above *s* a *redrawing* of *s* through *P* is shown.

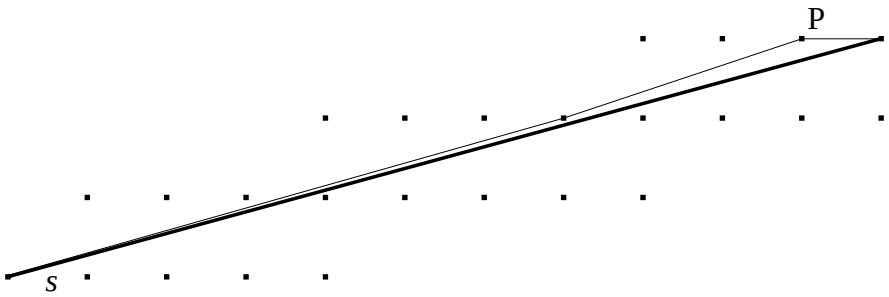


Figure 7

Intuitively, the process of redrawing can be understood as follows: Think of segment *s* as a rubber band and the points of the envelope as nails on a board. Now grip *s* at the true intersection point and pull it around *P*. The resulting polygonal path is the redrawing. The number of segments of this path is in the worst case logarithmic in the size of the grid, but it seems that in most cases only very few segments are created.

This approach guarantees that the polygonal line describing a segment always remains within the envelope of the original segment. We adopt the technique for realms. It then means that by redrawing a segment can never drift to the other side of a realm point. It might still happen, though, that after a redrawing a realm point is found to lie *on* a segment which it did not originally.

The formal definition of realm-based SDTs is organized as a series of layers. Each layer defines its own structures and primitives, using the notions of the layers below. We describe these layers bottom-up in the rest of this paper and the companion paper [GüS92]. Let us briefly provide an overview of this development.

Robust geometric primitives are introduced in Section 3 (and an appendix). This lowest layer defines a discrete space $N \times N$ where $N = \{0, ..., n-1\}$ is a subset of the natural numbers. The objects in this space are points and line segments with coordinates in N, called *N-points* and *N-segments*. A number of operations (predicates) such as whether an N-point lies *on* an N-segment or whether two N-segments *intersect*, and which N-point is the result of intersecting two N-segments, are defined. The crucial point is that these definitions are given in terms of error-free integer arithmetic, hence they are directly implementable.

In Section 4 geometric *realms* are defined as described above; elements are called *R-points* and *R-segments*. Basic operations on realms (given in Section 5) are insertion and deletion of N-points and N-segments. However, to cooperate with a database system, a realm – as an abstract data type – has a more general interface. For example, the operation of inserting an N-segment returns besides a modified realm a redrawing of the inserted segment and a set of redrawings of segments in the database that need to be modified together with logical pointers (SCIDs = spatial component identifiers) to database representations of these segments. The management of identifiers makes it necessary to offer operations that *register* spatial attribute values in the database with their underlying realm objects. A last group of operations allows to get a set of realm objects (within a rectangular window) and to identify objects when a close N-point is given. This makes it possible to display parts of a realm at a DBMS graphical user interface and to let a user build spatial attribute values by clicking at realm objects displayed.

The next layer (Section 6) defines certain structures present in a realm that serve as a basis for the definition of SDTs. A realm can be viewed as a planar graph; an *R-cycle* is a cycle of this graph. An *R-face* is an R-cycle possibly enclosing some other disjoint R-cycles corresponding to a region with holes. An *R-unit* is a minimal R-face. These three notions support the definition of a REGIONS data type. An *R-block* is a connected component of the realm graph; it supports the definition of a LINES data type. For all of these structures there are also predicates defined to describe their possible relationships.

After these preparations in the next layer (Section 7) the domains of spatial data types POINTS, LINES, and REGIONS are defined. A POINTS value is a set of R-points. There are two alternative views of LINES and REGIONS. The first is that a LINES value is a set of R-segments and a regions value a set of R-units. The other view is equivalent but “semantically richer”: A lines value is a set of disjoint R-blocks and a regions value a set of (edge-) disjoint R-faces. This completes the scope of this paper.

In the companion paper [GüS92] first the notion of *properties of collections of values* is introduced. The purpose is to support notions such as a *map* which is essentially a set of objects whose REGIONS attribute values are disjoint. One would like to identify such constraints and to use them for the definition of algebra operations, e.g. map overlay. Next, the *object model interface* (*OMI*) is defined. We identify a number of concepts that need to be present in the DBMS data (or object) model to allow it to cooperate with our spatial algebra. For example, the OMI must offer a function that yields for a

given object an associated SDT value, operations to form aggregates of objects or to group sets of objects by attribute values (which in connection with the spatial algebra allows to realize a “fusion” operation [ScV89, GaNT91]). The corresponding idea at the system level is that any extensible database system offering an OMI implementation can cooperate with a *spatial type extension package* (*STEP*) realizing the spatial algebra. Finally the spatial algebra itself is described; the semantics of all operations are formally defined. Due to the underlying realms, this algebra has simple and powerful operations, nice closure properties and observes a number of algebraic laws. It is straightforward to implement this algebra in such a way that the laws do also hold in practice.

3 Robust Geometric Primitives

The bottom layer of the formal development introduces a finite discrete space, points and line segments over this space, and some simple predicates and operations on them. Let $N = \{0, ..., n - 1\} \subseteq \mathbf{N}$. An *N-point* is a pair $(x, y) \in N \times N$. An *N-segment* is a pair of distinct *N*-points (p, q). P_N denotes the set of all *N*-points and S_N the set of all *N*-segments. We introduce the primitives shown in Figure 8 on *N*-points and *N*-segments. In this graph representation, undirected edges denote predicates and directed edges other operations (here *intersection* takes two *N*-segments and returns an *N*-point).

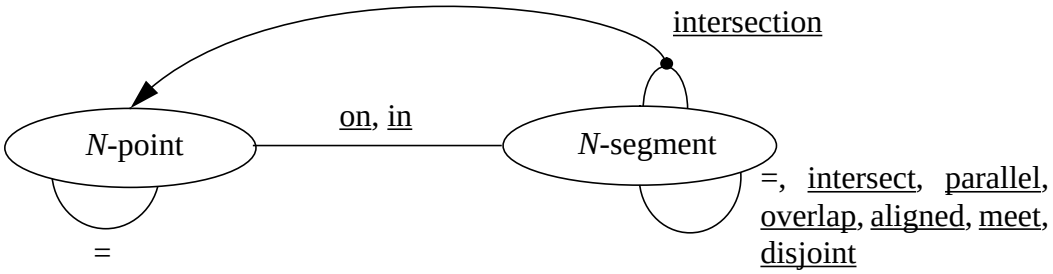


Figure 8

In the appendix these operations are defined in terms of integer arithmetic. Therefore they have a straightforward implementation free of numerical errors. We briefly explain them informally here: Two *N*-segments *meet* if they have exactly one end point in common. They *overlap* if they are collinear and share a (partial) *N*-segment. If they are collinear and do not share a (partial) *N*-segment, we call them *aligned*. If they have exactly one common point but do not meet, they *intersect*. If they have the same slope, they are *parallel*. They are *disjoint* if they are neither equal nor meet nor intersect. The *on* primitive tests if an *N*-point lies on an *N*-segment; the *in* primitive does nearly the same but the *N*-point must not coincide with one of the end points of the *N*-segment. The *intersection* primitive calculates the intersection point of two *N*-segments and rounds it to the nearest *N*-point.

4 Realms

Given *N*, a *realm over N*, or *N-realm* for short, is a set $R = P \cup S$ such that

(i) $P \subseteq P_N, S \subseteq S_N$

(ii) $\forall\, s \in S : s = (p, q) \Rightarrow p \in P \wedge q \in P$

(iii) $\forall\, p \in P\ \forall\, s \in S : \neg\, (p \text{ in } s)$

(iv) $\forall\, s, t \in S : \neg\, (s = t) \wedge \neg\, (s \text{ and } t \text{ intersect}) \wedge \neg\, (s \text{ and } t \text{ overlap})$

The elements of *P* and *S* are called *R-points* and *R-segments*. There is an obvious interpretation of a realm as a spatially embedded planar graph with set of nodes *P* and set of edges *S*.

5 Operations on Realms / The Realm Interface

Obviously the fundamental operations on realms are the insertion of an *N*-point or *N*-segment and the deletion of an *R*-point or *R*-segment. However, the interface is a bit more complex since we study realms not just as abstract entities but in connection with spatial databases. That means that there are spatial attribute values in the database depending on realm objects. This dependency needs to be modeled and treated by the operations.

Our approach to implement the dependency is the following: We assume that (the geometry of) a spatial attribute value of some object is stored together with the object in the database and there is a logical pointer from each segment or point describing the spatial value to the underlying realm object. Furthermore, associated with each realm object (point or segment) is a set of logical pointers; one pointer to each corresponding component of a spatial value in the database. In other words, points and segments in the database are doubly linked with the corresponding points and segments in the realm. Pointers from the database into the realm are realized by *realm object identifiers* from a set *ROID*, pointers from the realm into the database by *spatial component identifiers* from a set *SCID*. It is assumed that each *roid* or *scid* uniquely identifies the corresponding entity and that the implementation guarantees fast access to these components. Additionally, a *scid* also identifies the spatial value as a whole.

This approach stores geometries redundantly with a database object and in the realm. One might save some space by representing attribute values just by structures composed of pointers to realm objects. However, we believe its crucial for efficient query processing and well worth the extra space to keep the geometries with the objects. In this way one can directly apply SDT operations to spatial attribute values whereas otherwise it would always be necessary to access (load pages of) the underlying realm. Note that the two-way linking is necessary in any case since changes in the realm (e.g. a new intersection point on a segment) need to be propagated to the dependent spatial values.

We model the management of pointers formally as follows: Let *R* be a realm. A *representation of R* is a set of triples $\{(r, roid(r), scids(r)) \mid r \in R\}$ where *roid* is a function giving for a realm object its unique identifier in *ROID* and *scids* is a function returning the set of *SCIDs* of dependent components of spatial attribute values. We also allow the notation *roid*(*v*) to assign a new *ROID* to a newly created realm object *v*.

The realm interface is described by the following signature. Slightly extending standard notations we allow operators to return tuples of values and sets of values − the type of a set of X values is denoted by X*. The first group of operations are those mentioned above:

sorts *Realm, Point, Segment, RealmObject, ROID, SCID, Rectangle, Bool, Integer*

ops	*InsertNPoint*:	$Realm \times Point$	$\rightarrow$	$Realm \times ROID \times (SCID \times (Segment \times ROID)^*)^*$
	InsertNSegment:	$Realm \times Segment$	$\rightarrow$	$Realm \times (Segment \times ROID)^* \times (SCID \times (Segment \times ROID)^*)^* \times Bool$
	Delete:	$Realm \times ROID$	$\rightarrow$	$Realm \times Bool$

The sort (type) *Realm* refers to a realm representation, *Point* to the set P_N, *Segment* to the set S_N. *RealmObject* is a union type of *Point* and *Segment*; we assume one can recognize whether a given instance is a point or a segment. *ROID* and *SCID* have been discussed above. *Rectangle* denotes the set of axis-parallel rectangles definable over space $N \times N$, that is, *N-rectangles*.

The update operations implement the approach of Greene and Yao [GrY86] described in Section 2 to preserve the topology for a set of intersecting line segments when they are represented over a finite grid. Greene and Yao do not deal with collections of points that are part of our realms. We extend their approach by an additional integrity rule for points and line segments that are very close to each other. In Section 2 the concept of an *envelope* was already introduced as a set of grid points "adjacent" to a segment (formally defined in [GrY86]). Let us call the "proper envelope" the subset of envelope points that are not end points of the segment (denoted $\bar{E}(s)$ for segment s). Then the rule is:

No R-point lies on the proper envelope of any R-segment.

The intuition behind this is that points that are so close are meant to lie on the segment. Update operations maintain this constraint by redrawing the segment whenever a point is discovered to lie on its proper envelope (which can happen on point insertion or on segment insertion).

The operation *InsertNPoint* takes a realm and an *N*-point. It returns (i) the modified realm, (ii) an identifier for the inserted point, which could be an old one if the point was in the realm already, and (iii) a set of segments in the database that need to be redrawn, which may be empty. A segment may need redrawing because the point lies on its proper envelope. For each such segment its "address" in the database (SCID) together with a list of pairs (s, *roid*(s)) (where s is a segment of the redrawing) is returned. It is then the task of the DBMS to replace segments by their redrawings.

The operation *InsertNSegment* takes a realm and an *N*-segment. It returns (i) the modified realm, (ii) a list of segments with their *roids* which may contain either the original segment as the only element or a redrawing of this segment, (iii) a possibly empty set of segments that need to be redrawn (as for *InsertNPoint*). Here the inserted segment may need redrawing because it or its proper envelope touches *R*-points or because it intersects *R*-segments. The other segments need redrawing because they are intersected by this segment. The last parameter (iv) indicates whether insertion was performed; it was rejected, if not both end points of the segment were present in the realm.

The operation *Delete* takes a realm and the identifier of a realm object (point or segment) and removes the object from the realm if this doesn't violate certain integrity constraints. It returns (i) the modified realm and (ii) an indication whether the object was removed. The following conditions are checked: A point is only removed if there is no segment ending in the point. Any realm object is only removed if its set of *scids* (dependent objects) is empty.

The second group of operations supports the management of the two-way linking between realm objects and components of spatial values in the database:

Register:	*Realm* × *ROID* × *SCID*	→	*Realm*
Unregister:	*Realm* × *ROID* × *SCID*	→	*Realm*
GetSCIDs:	*Realm* × *ROID*	→	*SCID**
GetRealmObject:	*Realm* × *ROID*	→	*RealmObject*

Here *Register* informs a realm object *roid* about a spatial component *scid* depending on it. *Unregister* removes such an information. *GetSCIDs* returns the *scids* of spatial components depending on the given *roid*, *GetRealmObject* returns the geometry. These operations are to be used, for example, as follows: A spatial attribute value is constructed by selecting a number of realm objects in a certain order (this is supported by the last group of operations, see below). After all components have been selected, the representation of this value is built and stored in the database. Then all components are registered with their underlying realm objects. When a spatial attribute value is deleted, the registration is removed for all objects. *GetSCIDs* and *GetRealmObject* are general purpose operations to support query processing.

The last group of operations supports the selection of realm objects for the construction of spatial values:

Window:	$Realm \times Rectangle$	$\rightarrow$	$(RealmObject \times ROID)^*$
Identify:	$Realm \times Point \times Integer$	$\rightarrow$	$ROID \times Bool$

Window returns all realm objects together with their *roid* that are inside or intersect a given rectangular window. *Identify* tries to identify a realm object close to the *N*-point given as a parameter. The number given as a third parameter controls the “pick distance”. A *roid* (possibly undefined) is returned together with an indication whether identification was successful. These two operations can be used to retrieve a portion of a realm in order to define spatial attribute values over it. For example, this portion may be displayed at a graphical user interface. With a pointing device one can select *N*-points which through *Identify* determine realm objects from which the spatial values can be built.

We now define the semantics of the realm operations *InsertNPoint* and *InsertNSegment*. All the other operations are rather simple so that their meaning should be clear from the explanations above. *InsertNPoint* and *InsertNSegment* are described by giving algorithms for them. As mentioned before, the concepts of Greene and Yao [GrY86] are applied and extended, some of which need now to be explained in more detail. In [GrY86], for a set of line segments redrawings are computed in two phases. First, for each segment that needs to be redrawn, the set of points on its envelope, that need to be passed by the redrawing, is computed. These points are attached to the segment in the form of “hooks”. A hook is a short directed line segment (an “arrow”) from a point on the segment to the envelope point that needs to be passed.

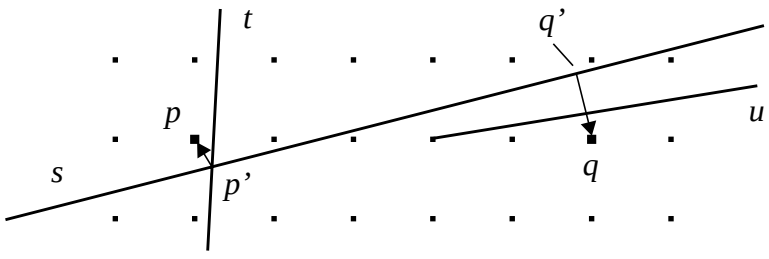


Figure 9

Figure 9 shows a segment *s* with two hooks <*p*’, *p*>and <*q*’, *q*>. Such a “hooked segment” is represented as a list (*s*; <*p*’, *p*>, <*q*’, *q*>). The first hook arises from the intersection of segments *s* and *t* in *p*’; when this intersection is discovered, the hook <*p*’, *p*> is added to *both* segments *s* and *t*. Let us assume that the other hook in Figure 9 arises from the fact that point *q* was inserted into the realm;

since it lies on the envelope of *s*, *s* should go through *q*. In such a case we take the point on the segment closest to the target point (in this case *q*') as the start point of the hook and denote it as *base*(<target>,), in this case *q*' = *base*(*q*, *s*).[4]

Figure 9 also illustrates that for each hook created one generally needs to check whether any segments are intersected by it. In the example an intersection with segment *u* would be discovered and a corresponding hook <*q*", *q*> be added to segment *u* (where *q*" is the intersection point of the hook and *u*). Only after all hooks have been determined, redrawings (polygonal lines within the envelope) are computed. This can now be done for each hooked segment independently from all other segments. See [GrY86] for a description of how redrawings are computed.

The algorithm *InsertNPoint* has to treat the following cases: (i) the point is already present in the realm, (ii) the point is new and does not lie on any envelope, and (iii) the point falls on one or more proper envelopes. Only the last case is a bit more complex: All segments whose envelopes are touched get a hook. Later, all those segments are redrawn. The description uses predicates *ExistsRPoint* and *ExistsRSegment* with the obvious meaning to check whether a realm object to be created is present already.

algorithm *InsertNPoint* (*R*, *p*, *R*', *r*, *SP*)
{Inputs are a realm $R = P \cup S$ and an *N*-point *p*. Outputs are the modified realm *R*', a realm object identifier *r* for *p*, and a set *SP* of spatial component identifiers and redrawings for the spatial objects which have to be updated.}

Step 1
SP := ∅;
if $\exists\, q \in P : p = q$ (at most one such *R*-point can exist)
then *r* := *roid*(*q*); *R*' := *R*
else **if** $\forall\, s \in S : p \notin \bar{E}\,(s)$
then *r* := *roid*(*p*); $R' := R \cup \{(p, r, \emptyset)\}$
else (*R*-segments exist whose proper envelopes contain *p*)
$S_{rd} := \emptyset$; (a set of *R*-segments which have to be redrawn)
r := *roid*(*p*);
$S_{env} := \{s \in S \mid p \in \bar{E}\,(s)\}$;
for each *s* **in** S_{env} **do**
Insert a hook *h* = <*base* (*p*, *s*), *p*> on *s*; $S_{rd} := S_{rd} \cup \{s\}$
(one does not need to check for segments in *S* intersecting *h* because they are already in S_{env})

Step 2 (redraw hooked lines)
Let $S_{rd} := \{t_1, ..., t_n\}$. Let $\{t_{i,1}, ..., t_{i,k_i}\}$ be the set of k_i *R*-segments of the redrawing of t_i through *p*.
Let $t_{i,j} = (p_{i,j}, q_{i,j})$, $i \in \{1, ..., n\}$, $j \in \{1, ..., k_i\}$

Step 3 (update realm)
$R' := R \setminus \{(t_i, roid(t_i), scids(t_i)) \mid i \in \{1, ..., n\}\}$

(Insert the end points of the *R*-segments of the redrawings and the *R*-segments themselves if they do not already exist in the realm)

[4] In [GrY86] all hooks arise from segment intersections.

for each i **in** 1..n **do**
for each j **in** 1..k_i **do**
if not *ExistsRPoint*($p_{i,j}$) **then** $R' := R' \cup \{(p_{i,j}, roid(p_{i,j}), \emptyset)\}$;
if not *ExistsRPoint*($q_{i,j}$) **then** $R' := R' \cup \{(q_{i,j}, roid(q_{i,j}), \emptyset)\}$;
if not *ExistsRSegment*($t_{i,j}$) **then** $R' := R' \cup \{(t_{i,j}, roid(t_{i,j}), \emptyset)\}$;

$$SP := \bigcup_{i=1}^{n} \{(sc, \{(t_{i,j}, roid(t_{i,j})) \mid j \in \{1, ..., k_i\}\}) \mid sc \in scids(t_i)\}$$

end *InsertNPoint*.

The algorithm *InsertNSegment* first checks whether the end points of the segment are present in the realm; otherwise it rejects insertion. This agrees with the graph-theoretic view of a realm: An edge can only exist if its nodes are there. It implies that the user of the realm layer has to make sure that the points are present (in case of doubt just insert them first; this doesn't hurt). Hence, when a segment is inserted, it is known that the end points have already interacted properly with envelopes of other segments. The following cases are now distinguished: (i) the segment is in the realm already, (ii) the segment is new and doesn't touch anything, and (iii) the segment may intersect some other segments and / or its envelope touches some realm points.

algorithm *InsertNSegment* (*R, s, R', RD, SP, ok*)
{Inputs are a realm $R = P \cup S$ and an *N*-segment *s*. Outputs are the modified realm *R'*, a set *RD* of pairs of *R*-segments and realm object identifiers either for *s* or a redrawing of *s*, a set *SP* of spatial component identifiers and redrawings for the spatial objects which have to be updated, and a parameter *ok* which indicates whether the insertion was performed. Insertion was rejected if the end points of *s* were not present in the realm.}

Step 1 (initializations)
$SP := \emptyset$;

Step 2 (check, whether end points of *s* have already been inserted)
Let $s = (q_1, q_2)$. *ok* := *ExistsRPoint*(q_1) **and** *ExistsRPoint*(q_2);

if *ok* **then** (execute steps 3-5)

Step 3 (insert hooks)
if $\exists\, t \in S : s = t$ (at most one such *R*-segment can exist)
then $R' := R$; $RD := \{(s, roid(t))\}$
else **if** $\forall\, t \in S : s$ and t are <u>disjoint</u> $\wedge\ \forall\, p \in P : p \notin \bar{E}(s)$
then $R' := R \cup \{(s, roid(s), \emptyset)\}$; $RD := \{(s, roid(s))\}$
else (*s* intersects *R*-segments and/or there are *R*-points in the proper envelope of *s*)
$S_{rd} := \emptyset$ (the set of *R*-segments which have to be redrawn)
(Get all *R*-points lying in the proper envelope of *s* except for the end points. Get all *R*-segments intersecting *s*.)
$P_{env}(s) := \{p \in P \mid p \in \bar{E}(s)\}$
$S_{intersect}(s) := \{t \in S \mid s$ and t <u>intersect</u>$\}$
for each p **in** $P_{env}(s)$ **do**
Insert a hook $h = <base(p, s), p>$ on s
for each t **in** S **do**
if h and t intersect at p'
then
Insert a hook $h = <p', p>$ on t; $S_{rd} := S_{rd} \cup \{t\}$

for each t **in** $S_{intersect}(s)$ **do**
Insert a hook $h = <q, p>$ both on s and on t from the intersection point q of s and t to the closest grid point p (Note that $p = q$ is possible if q is a grid point)
$S_{rd} := S_{rd} \cup \{t\}$
for each v **in** S **do**
if h and v intersect at p'
then
Insert a hook $h' = <p', p>$ on v; $S_{rd} := S_{rd} \cup \{v\}$

Step 4 (redraw hooked lines)
Redraw s. Let $\{s_1, ..., s_m\}$ be the R-segments of the redrawing of s. Let $s_i = (p_i, q_i)$, $i \in \{1, ..., m\}$. Redraw all R-segments of $S_{rd} := \{t_1, ..., t_n\}$. Let $\{t_{i,1}, ..., t_{i,k}\}$ be the set of k_i R-segments of the redrawing of t_i. Let $t_{i,j} = (p_{i,j}, q_{i,j})$, $i \in \{1, ..., n\}$, $j \in \{1, ..., k_i\}$

Step 5 (update realm)
$R' := R \setminus \{(t_i, roid(t_i), scids(t_i)) \mid i \in \{1, ..., n\}\}$

(Insert the end points of the R-segments of the redrawings and the R-segments themselves if they do not already exist in the realm.)
for each i **in** $1..m$ **do**
if not $ExistsRPoint(p_i)$ **then** $R' := R' \cup \{(p_i, roid(p_i), \emptyset)\}$
if not $ExistsRPoint(q_i)$ **then** $R' := R' \cup \{(q_i, roid(q_i), \emptyset)\}$
if not $ExistsRSegment(s_i)$ **then** $R' := R' \cup \{(s_i, roid(s_i), \emptyset)\}$

for each i **in** $1..n$ **do**
for each j **in** $1..k_i$ **do**
if not $ExistsRPoint(p_{i,j})$ **then** $R' := R' \cup \{(p_{i,j}, roid(p_{i,j}), \emptyset)\}$
if not $ExistsRPoint(q_{i,j})$ **then** $R' := R' \cup \{(q_{i,j}, roid(q_{i,j}), \emptyset)\}$
if not $ExistsRSegment(t_{i,j})$ **then** $R' := R' \cup \{(t_{i,j}, roid(t_{i,j}), \emptyset)\}$

$RD := \{(s_i, roid(s_i)) \mid i \in \{1, ..., m\}\}$
$SP := \bigcup_{i=1}^{n} \{(sc, \{(t_{i,j}, roid(t_{i,j})) \mid j \in \{1, ..., k_i\}\}) \mid sc \in scids(t_i)\}$

end *InsertNSegment*.

6 Realm-Based Structures and Primitives

We can now assume that the problems of numerical robustness and topological correctness are solved by the lower layers. Given is a realm which can be viewed as a planar graph over the grid $N \times N$. Within a realm one can discover certain structures and relationships between these structures useful for the definition of spatial data types. These structures are called *R-cycle*, *R-face*, *R-unit*, and *R-block*. For the relationships we will define a number of predicates (primitives).

An *R-cycle c* is just a cycle in the graph interpretation of a realm, defined by a set of *R*-segments $S(c) = \{s_1, ..., s_m\}$, such that

(i) $\forall\, i \in \{1, ..., m\} : s_i$ meets $s_{(i+1) \bmod m}$

(ii) No other pairs of segments in $S(c)$ meet.

Obviously the following relationships may exist between an *N*-point *p* and an *R*-cycle *c*:

(i) p *on* $c \quad :\Leftrightarrow \quad \exists\, s \in S(c) : p$ on s

For $p = (x, y)$ let $s_p = ((x, y), (x, n-1))$ (that is, a vertical segment extending from p upwards to the edge of the grid). Let $S_r(c)$ be the segments in $S(c)$ whose right end point, but not the left one, is on s_p. Let $S_i(c)$ be the segments in $S(c)$ that intersect s_p. Then

(ii) p *in* $c \quad :\Leftrightarrow \quad \neg\, p$ *on* $c \wedge |S_r(c)| + |S_i(c)|$ is odd[5]

(iii) p *out* $c \quad :\Leftrightarrow \quad \neg\, (p$ *on* $c \vee p$ *in* $c)$

Hence c partitions the set P_N into three subsets $P_{in}(c)$, $P_{on}(c)$, and $P_{out}(c)$. Let $P(c) := P_{on}(c) \cup P_{in}(c)$.

Cycles are interesting because they are the basic entities for the definition of regions over realms. The relationships shown in Figure 10 may be distinguished between two R-cycles c_1 and c_2 :

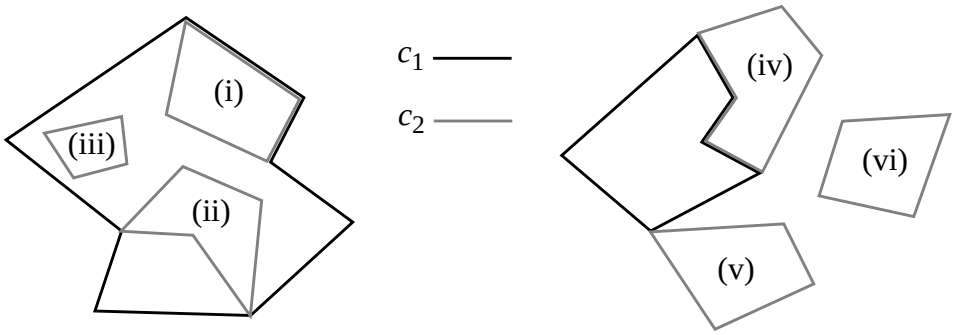


Figure 10

We introduce the following terminology for these configurations:

c_2 is

- (*area-*)*inside* (i, ii, iii)
- *edge-inside* (ii, iii)
- *vertex-inside* (iii)

c_1 and c_2 are

- *area-disjoint* (iv, v, vi)
- *edge-disjoint* (v, vi)
- (*vertex-*)*disjoint* (vi)

c_1.

The meaning is that (i) c_2 is (w.r.t *area*) *inside* c_1, (ii) additionally has no common edges with c_1, (iii) has not even common vertices with c_1. Similarly (iv) c_2 is *disjoint* w.r.t. *area* with c_1, (v) additionally has no common edges with c_1, (vi) additionally has not even common vertices with c_1. *area-inside* is the standard interpretation of the term *inside*, *vertex-disjoint* the standard interpretation of the term *disjoint*.

Furthermore there are two positive notions: c_1 and c_2 are *adjacent* if they are area-disjoint and have common edges, they *meet* if they are area-disjoint and have common vertices. These predicates are formally defined as follows:

c_1 (*area-*)*inside* $c_2 \quad :\Leftrightarrow \quad P(c_1) \subseteq P(c_2)$

c_1 *edge-inside* $c_2 \quad :\Leftrightarrow \quad c_1$ *area-inside* $c_2 \wedge S(c_1) \cap S(c_2) = \varnothing$

c_1 *vertex-inside* $c_2 \quad :\Leftrightarrow \quad c_1$ *edge-inside* $c_2 \wedge P_{on}(c_1) \cap P_{on}(c_2) = \varnothing$

c_1 and c_2 are *area-disjoint* $\quad :\Leftrightarrow \quad P_{in}(c_1) \cap P(c_2) = \varnothing \wedge P_{in}(c_2) \cap P(c_1) = \varnothing$

[5] This is a precise grid-based formulation of the well-known “plumbline” algorithm.

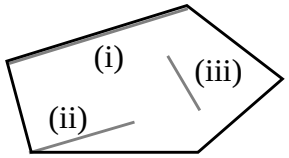
c_1 and c_2 are *edge-disjoint* $:\Leftrightarrow$ c_1 and c_2 are *area-disjoint* $\wedge\ S(c_1) \cap S(c_2) = \varnothing$

c_1 and c_2 are (*vertex-*)*disjoint* $:\Leftrightarrow$ c_1 and c_2 are *edge-disjoint* $\wedge\ P_{on}(c_1) \cap P_{on}(c_2) = \varnothing$
(which is equivalent to saying that $P(c_1) \cap P(c_2) = \varnothing$)

c_1 and c_2 are *adjacent* $:\Leftrightarrow$ c_1 and c_2 are *area-disjoint* $\wedge\ S(c_1) \cap S(c_2) \neq \varnothing$

c_1 and c_2 *meet* $:\Leftrightarrow$ c_1 and c_2 are *edge-disjoint* $\wedge\ P_{on}(c_1) \cap P_{on}(c_2) \neq \varnothing$

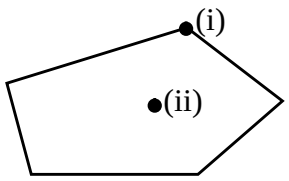
One can observe similar ways how an *R*-segment *s* can lie within an *R*-cycle *c*:



- *s* (*area-*)*inside* *c* (i, ii, iii)
- *s* *edge-inside* *c* (ii, iii)
- *s* *vertex-inside* *c* (iii)

Figure 11

For an *R*-point *p* and an *R*-cycle *c* we have two possibilities:



- *p* (*area-*)*inside* *c* (i, ii)
- *p* *vertex-inside* *c* (ii)

Figure 12

Formal definitions are left to the reader.

Based on the concept of *R*-cycles, for the definition of a spatial data type for regions the notions *R-face* and *R-unit* are introduced which describe regions from two different perspectives and which are used equivalently. Both of them essentially define polygonal regions with holes. An *R*-unit is a “minimal” *R*-face in the sense that any *R*-face within the *R*-unit is equal to the *R*-unit. Hence *R*-units are the smallest region entities that exist over a realm. We will see that any two *R*-units are area-disjoint and that any *R*-face can be described as a set of *R*-units. In the next section a region (data type) will be defined that can either be viewed as a set of *R*-faces or, equivalently, as a set of *R*-units. The first view emphasizes a minimal representation of the boundary of a region whereas the latter view supports the definition of set operations for regions. We will define operations to convert between the two (formal) representations.

Let $C(R)$ denote the set of all *R*-cycles. An *R-face* f is a pair (c, H) where c is an *R*-cycle and $H = \{h_1, ..., h_m\}$ is a (possibly empty) set of *R*-cycles such that the following conditions hold (let $S(f)$ denote the set of all segments of all cycles of f):

(i) $\forall\ i \in \{1, ..., m\} : h_i$ *edge-inside* c

(ii) $\forall\ i, j \in \{1, ..., m\}, i \neq j : h_i$ and h_j are *edge-disjoint*

(iii) Each cycle in $S(f)$ is either equal to c or to one of the cycles in H (no other cycle can be formed from the segments of f)

The last condition ensures uniqueness of representation, that is, there are no two different

interpretations of a set of segments as sets of faces. For example, it guarantees that the configuration shown in Figure 13 must be interpreted as two faces, and not as a single face with 5 holes (since under the latter interpretation the cycle drawn fat would violate condition (iii)).

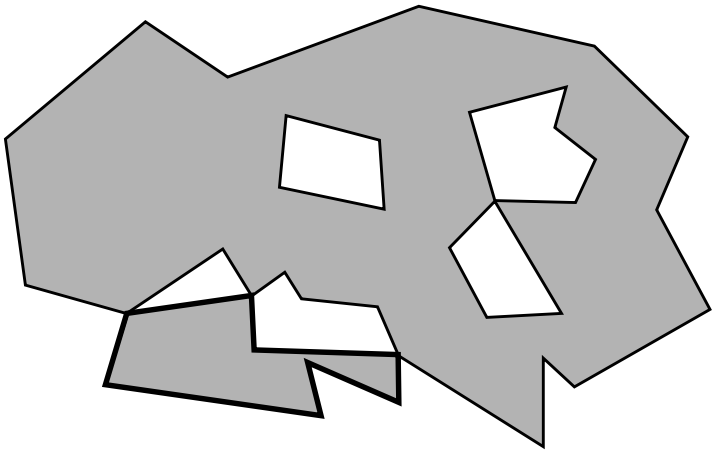


Figure 13

With terms defined below condition (iii) can be rephrased as “an *R*-face cannot be decomposed into two or more edge-disjoint *R*-faces”.

The grid points belonging to an *R*-face *f* are defined as:

$$P(f) := P(c) \setminus \bigcup_{i=1}^{m} P_{\text{in}}(h_i)$$

Let $S(F)$ denote the set of all *R*-segments of a set of *R*-faces *F*.

The possible relationships between an *R*-point *p* or an *R*-segment *s* and an *R*-face $f = (c, H)$ are:

(i) p (*area-*)*inside* $f \;:\Leftrightarrow\; p$ *area-inside* $c \wedge \forall\, h \in H : \neg\, p$ *vertex-inside* h

(ii) s (*area-*)*inside* $f \;:\Leftrightarrow\; s$ *area-inside* $c \wedge \forall\, h \in H : \neg\, s$ *edge-inside* h

The various notions of *inside* and *disjoint* can be extended for the comparison of two *R*-faces $f = (f_0, \bar{F})$ and $g = (g_0, \bar{G})$, for example:

$$f \text{ (area-)inside } g \;:\Leftrightarrow\; f_0 \text{ area-inside } g_0 \wedge \forall\, \bar{g} \in \bar{G} : \bar{g} \text{ area-disjoint } f_0 \vee \exists\, \bar{f} \in \bar{F} : \bar{g} \text{ area-inside } \bar{f}$$

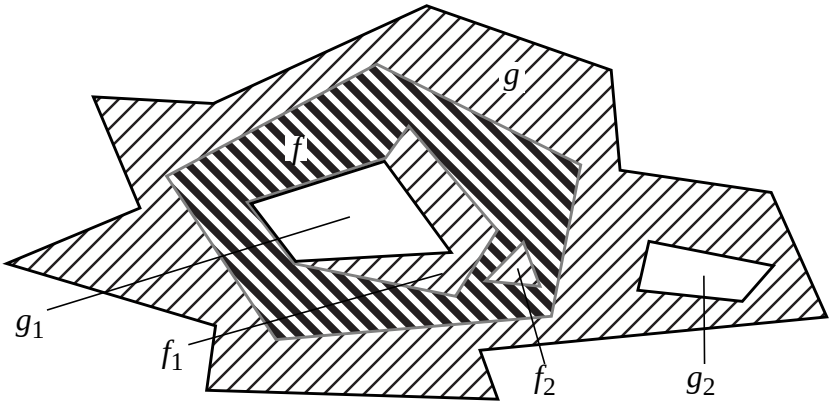


Figure 14

This definition is illustrated in Figure 14.

f *area-disjoint* $g \quad :\Leftrightarrow f_0$ *area-disjoint* $g_0 \vee \exists\, \bar{g} \in \overline{G} : f_0$ *area-inside* $\bar{g} \vee \exists\, \check{f} \in \overline{F} : g_0$ *area-inside* $\check{f}$

f *edge-disjoint* $g \quad :\Leftrightarrow f_0$ *edge-disjoint* $g_0 \vee \exists\, \bar{g} \in \overline{G} : f_0$ *edge-inside* $\bar{g} \vee \exists\, \check{f} \in \overline{F} : g_0$ *edge-inside* $\check{f}$

The meaning of the remaining predicates *edge-inside*, *vertex-inside*, *vertex-disjoint*, *adjacent*, *meet* should be clear; definitions are omitted for brevity.

An *R*-unit as a minimal *R*-face is defined as follows. Let $F(R)$ denote the set of all possible *R*-faces. Let f be an *R*-face.

f is an *R-unit* $\quad :\Leftrightarrow \quad \forall\, g \in F(R) : g$ *area-inside* $f \quad \Rightarrow \quad g = f$

We also denote by $U(R)$ the set of all *R*-units.

Our goal is now to establish an equivalence between two representations of a region over a realm, namely, as a set of (pairwise) edge-disjoint *R*-faces, and as a set of *R*-units. First we consider the conversion of a set of faces into a set of units. We need two lemmas, whose proofs are technical and are only sketched:

Lemma 6-1 Let f be an *R*-face and u an *R*-unit. Then either u *area-inside* f or u *area-disjoint* f.

The idea of the proof is that if this is not the case, then one of the cycles of f, say f', must properly intersect one of the cycles of u, say u'.

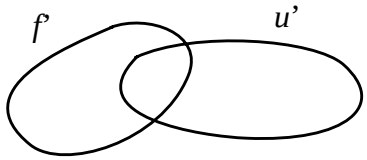


Figure 15

But then a part of f' lies within u and forms a cycle there with a part of u'. Hence there would be a face contained in u different from u which contradicts the definition of an *R*-unit.

Lemma 6-2 Let f be an *R*-face and u an *R*-unit such that u *area-inside* f. Then "subtracting" u from f results in a set of *R*-faces.

The idea of the proof is the following: If u is even *edge-inside* f then removing the area of u from f just adds another hole to f. If u's outer cycle u_0 has some adjacent parts with f's outer cycle f_0, then a

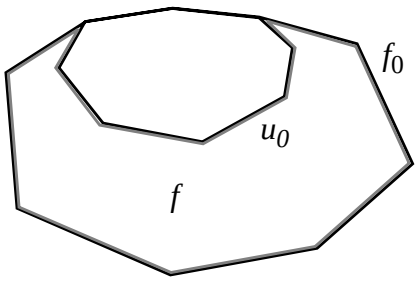


Figure 16

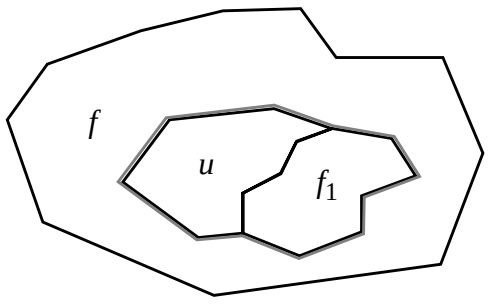


Figure 17

"bay" is formed in f_0 (Figure 16). If it is adjacent with a hole f_1 in f, then f_1 will grow (Figure 17). If several adjacencies are present, then f may be decomposed into several faces.

The second lemma implies that the units inside a face *f* cover the area of *f* completely. For, if some area were left, it would form its own face which could again be decomposed into units.

Therefore the following definition correctly decomposes faces into units. Let *F* be a set of edge-disjoint *R*-faces.

$units(F) := \{u \in U(R) \mid \exists f \in F: u \text{ area-inside } f\}$

We now consider the conversion of a set of units into a set of faces. Given a set of *R*-segments *S*, we say, *S describes a set of pairwise edge-disjoint R-faces* :⇔ there exists a set of edge-disjoint *R*-faces *F* such that $S = S(F)$. Furthermore, let Δ denote the operator for symmetrical set difference, that is, $V \Delta W = (V \setminus W) \cup (W \setminus V)$. Δ forms the union of two sets removing their intersection. The operator is associative and commutative. The basis for the conversion is the following lemma:

Lemma 6-3 Let *f* and *g* be two area-disjoint *R*-faces. Then $S(f) \Delta S(g)$ describes a set of edge-disjoint *R*-faces.

The basic idea is that the Δ operator just removes segments that are common to both faces. The area-disjointness condition makes sure that only boundaries between adjacent areas are removed (and not boundaries between a covered region in one face and a hole in the other face).

The lemma can be extended to two sets of faces: Let *F, G* be two sets of edge-disjoint *R*-faces such that the faces in $F \cup G$ are pairwise area-disjoint. Then $S(F) \Delta S(G)$ describes a set of edge-disjoint *R*-faces. Let the resulting set of *R*-faces be denoted as $F + G$. Now the conversion from units to faces can be defined as follows. Let *U* be a set of *R*-units.

$$faces(U) = \sum_{u \in U} \{u\}$$

We summarise the equivalence in

Theorem 6-4 $\forall F \subseteq F(R)$: $faces(units(F)) = F$

For the definition of a spatial data type for lines the notion of an *R-block* is introduced. A set *S* of *R*-segments is called *connected* :⇔ $\forall r, t \in S \; \exists s_1, ..., s_m, s_i \in S : r = s_1, t = s_m$, and $\forall i \in \{1, ..., m - 1\}$: s_i and s_{i+1} meet. An *R-block* is a connected set of *R*-segments. Two *R*-blocks b_1 and b_2 are *disjoint* :⇔ $\forall s_1 \in S(b_1) \; \forall s_2 \in S(b_2)$: s_1 and s_2 are disjoint. For an *R*-point *p* we consider the angularly sorted cyclic list L_p of *R*-segments $s \in S(b_1) \cup S(b_2)$ that meet in *p*. *p* is called a *meeting point* if L_p can be subdivided into two sublists $L_{p,1}$ and $L_{p,2}$ (whose concatenation leads to L_p) so that all *R*-segments of $L_{p,1}$ are elements of $S(b_1)$ and all *R*-segments of $L_{p,2}$ are elements of $S(b_2)$, or vice versa (see Figure 18).

Figure 18: *p* is a meeting point, *p*' is not a meeting point.

Two R-blocks b_1 and b_2 *meet* :⇔

(i) $\forall\, s \in S(b_1)\ \forall\, t \in S(b_2) : s \neq t$

(ii) $\forall\, s \in S(b_1)\ \forall\, t \in S(b_2) : s$ and t meet $\Rightarrow s$ and t meet in a meeting point.

Again, we have two equivalent representations of a lines value, namely, as a set of segments, or as a set of disjoint R-blocks. For a set of segments $S' \subseteq S$ let *blocks*(S') denote its partition into maximal connected components. Then S(*blocks*(S')) = S'.

7 Realm-Based Spatial Data Types

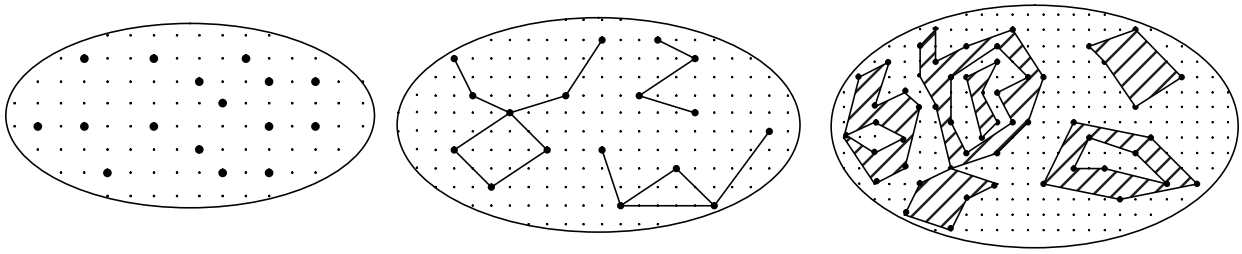
A formal definition of realm-based spatial data types (including operations) is given in the companion paper [GüS92]. However, to conclude this paper we show the connection between the realm-based structures of the previous section and the domains of the spatial data types. The basic types introduced are called POINTS, LINES, and REGIONS. There is a "flat" and a "structured" view of values of these types. The "flat" view is the following:

> *For a given realm R, a value of type POINTS is a set of R-points, a value of type LINES is a set of R-segments, and a value of type REGIONS is a set of R-units.*

The structured view, that we shall assume as the formal definition, is as follows:

> *For a given realm R, a value of type POINTS is a set of R-points, a value of type LINES is a set of pairwise disjoint R-blocks, and a REGIONS value is a set of pairwise edge-disjoint R-faces.*

We have shown in the previous section that the two views are equivalent. The first view is conceptually very simple and supports a direct understanding of set operations. The second view shows LINES and REGIONS values as consisting of a number of *components* (blocks or faces), allows one to express relationships between these components and also emphasizes the representation of the boundary in case of regions. Figure 19 illustrates the data types.



a POINTS value a LINES value a REGIONS value

Figure 19

It should be obvious now that these data types have very nice closure properties. For example, the set operations are defined as follows. Let P_1 and P_2 be two POINTS values, L_1 and L_2 two LINES values, and R_1, R_2 two REGIONS values. Then

union (P_1, P_2) := $P_1 \cup P_2$
union (L_1, L_2) := *blocks*($S(L_1) \cup S(L_2)$)
union (R_1, R_2) := *faces*(*units*(R_1) ∪ *units*(R_2))

For ***intersection*** and ***difference*** the definitions are analogous. The primitives introduced in the previous sections offer a formal basis for the definition of operations of a spatial algebra. For example, one can define what it means for two regions to be adjacent:

F and G are ***adjacent*** $:\Leftrightarrow$ F and G are ***area-disjoint*** $\wedge\ \exists f \in F\ \exists g \in G$: f and g are *adjacent*

(assuming ***area-disjoint*** to be defined already for regions).

8 Conclusions

In this paper we have offered geometric realms as a concept to solve several problems related to spatial data types for database systems. In particular, realms solve the geometric consistency problem as well as problems of numeric robustness and topological correctness. Realm-based structures can be used for the definition of quite general spatial data types and guarantee all the desired closure properties in theory as well as in computational practice. Starting from integer arithmetics, we have developed bottom-up a precise formal framework that makes it easy to define spatial algebras and to implement them correctly. Indeed, such a realm-based algebra is defined in [GüS92].

In closing, let us briefly discuss some open problems and questions that arise with this approach.

Topological correctness. Although it goes a long way, the approach of Greene and Yao does not completely guarantee topological correctness. As is also stated in [GrY86], through the finite representation "... disjoint points and lines may collapse. However, aside from such degeneracies, we do guarantee that topology does not change." There has been a lot of work on numeric robustness and topological correctness for geometric computation (e.g. [OtTU87, GuiSS89, EdM88, NaME90]). We have selected [GrY86] because it fits well with our idea of realms as grid-based planar graphs underlying spatial data types. However, one might try to extend this by adding further integrity constraints (such as our rule that R-points must not lie on envelopes) or by techniques from the other approaches (e.g. symbolic reasoning) to avoid the remaining anomalies.

Efficiency. The realm update algorithms of Section 5 have been given in rather abstract terms, one might be concerned, whether they are efficiently implementable. We suggest to represent a realm in a spatial index structure (e.g. [Gut84]) and are currently implementing realms on the basis of LSD-Trees [HeSW89]. The lookup operations needed in the algorithms can then be performed efficiently. Such a realm representation can at the same time be used as an index into the database. A separate issue is the efficiency of spatial algebra operations (such as intersecting two regions). This can be done by variations of plane-sweep algorithms such as [NiP82, BeO79]. Indeed, these algorithms are now much simpler and more efficient since they do not need to discover new intersections and do not have to treat special cases. The study of algorithms for realm-based data types might become an interesting field of its own.

Space overhead. By redrawing, many more segments may be created than were present in the original set of intersecting line segments. How many more, is an interesting question that should be studied theoretically as well as in experiments with "real life" data. In any case, we feel one cannot trade correctness for space.

Multiple realms. In this paper we have only discussed the case of a single realm underlying all spatial data (of a certain application area). There are several reasons why one might be interested in several

realms over the same area. One is to reduce space overhead (by not intersecting spatial values of different realms). Another reason is that there exist interesting SDT operations that are not closed with respect to a realm. An example is the creation of a “buffer area” around a polyline. To accommodate such operations one might dynamically create a realm containing just the “new” spatial values, select a set of SDT values that might interact with these new geometries and create a “small” realm for them, and then use a “merge” operation on realms to compute all intersections correctly.

Appendix: Definition of Geometric Primitives

As a basis for definition and implementation we only assume that the following arithmetic primitives are available and are error-free with respect to overflow.

$$\text{INT} \times \text{INT} \to \text{INT} \qquad +, -, *, \textbf{div}, \textbf{mod}$$
$$\text{INT} \times \text{INT} \to \text{BOOL} \qquad =, \neq, <, \leq, \geq, >$$

To fulfill this requirement in an implementation it is sufficient that the INT data type can represent numbers in the range $[-2n^3, 2n^3]$ where $n = |N|$ (see below). Either this relationship holds between N needed by the application and a programming language integer type, or one needs to implement a special integer type with these operations.

For an N-point p we denote by $p.x$ and $p.y$ its first and second component, respectively. Two N-points p and q are equal,

$$p = q \quad :\Leftrightarrow \quad p.x = q.x \wedge p.y = q.y \; .$$

Two N-segments $s_1 = (p_1, p_2)$ and $s_2 = (q_1, q_2)$ are equal,

$$s_1 = s_2 \quad :\Leftrightarrow \quad (p_1 = q_1 \wedge p_2 = q_2) \vee (p_1 = q_2 \wedge p_2 = q_1)$$

Let $s_1 = (p_1, p_2) = ((x_{11}, y_{11}), (x_{12}, y_{12}))$ and $s_2 = (q_1, q_2) = ((x_{21}, y_{21}), (x_{22}, y_{22}))$ be two N-segments. For the calculation of a possible intersection point of the two N-segments we use the following matrix representation where λ, μ are rational numbers (to be represented by pairs of INT values).

$$\begin{bmatrix} x_{11} \\ y_{11} \end{bmatrix} + \lambda \left(\begin{bmatrix} x_{12} \\ y_{12} \end{bmatrix} - \begin{bmatrix} x_{11} \\ y_{11} \end{bmatrix} \right) = \begin{bmatrix} x_{21} \\ y_{21} \end{bmatrix} + \mu \left(\begin{bmatrix} x_{22} \\ y_{22} \end{bmatrix} - \begin{bmatrix} x_{21} \\ y_{21} \end{bmatrix} \right)$$

This leads to the following inhomogeneous linear equation system in two variables:

$$x_{11} - x_{21} = -\lambda\,(x_{12} - x_{11}) + \mu\,(x_{22} - x_{21})$$
$$y_{11} - y_{21} = -\lambda\,(y_{12} - y_{11}) + \mu\,(y_{22} - y_{21})$$

Let $a_{11} := x_{11} - x_{12}$, $a_{12} := x_{22} - x_{21}$, $b_1 := x_{11} - x_{21}$, $a_{21} := y_{11} - y_{12}$, $a_{22} := y_{22} - y_{21}$, and $b_2 := y_{11} - y_{21}$. Then

$$\begin{aligned} a_{11}\,\lambda + a_{12}\,\mu &= b_1 \\ a_{21}\,\lambda + a_{22}\,\mu &= b_2 \end{aligned} \quad \Rightarrow \quad \begin{aligned} \lambda\,(a_{11}\,a_{22} - a_{12}\,a_{21}) &= b_1\,a_{22} - b_2\,a_{12} \\ \mu\,(a_{11}\,a_{22} - a_{12}\,a_{21}) &= b_2\,a_{11} - b_1\,a_{21} \end{aligned}$$

With $D := a_{11}\,a_{22} - a_{12}\,a_{21}$, $D_1 := b_1\,a_{22} - b_2\,a_{12}$, $D_2 := b_2\,a_{11} - b_1\,a_{21}$, and $D \neq 0$ we get

$$\lambda = \frac{D_1}{D}, \; \mu = \frac{D_2}{D}. \qquad \text{(EQ 1)}$$

Two *N*-segments intersect if $D \neq 0$ and $0 < \lambda < 1$ and $0 < \mu < 1$. Note that the situation where an end point of one segment lies on the other segment is excluded. In particular no two end points are equal. Two *N*-segments are parallel if $D = 0$.

For an *N*-segment $s = ((x_1, y_1), (x_2, y_2))$, $x\text{-}ext(s) := \{min(x_1, x_2), ..., max(x_1, x_2)\} \subseteq N$ and $y\text{-}ext(s) := \{min(y_1, y_2), ..., max(y_1, y_2)\} \subseteq N$ denote the *x*- and *y*-intervals of its bounding box. The resulting intervals are called *N-intervals*. Two *N*-intervals I_1 and I_2 *overlap* if $card(I_1 \cap I_2) > 1$. They are *disjoint* if $I_1 \cap I_2 = \varnothing$. Two *N*-segments s_1, s_2 overlap if

(i) $D = 0$

(ii) $D_1 = D_2 = 0$

(iii) $x\text{-}ext(s_1)$ and $x\text{-}ext(s_2)$ overlap $\vee$ $y\text{-}ext(s_1)$ and $y\text{-}ext(s_2)$ overlap.

If condition (iii) does not hold and the *x*- and *y*-intervals of s_1 and s_2 are disjoint, the two *N*-segments are called aligned. Two *N*-segments $s_1 = (p_1, p_2)$ and $s_2 = (q_1, q_2)$ meet if they have exactly one end point in common. Two *N*-segments are disjoint if they are neither equal nor meet nor intersect. If two *N*-segments $s_1 = (p_1, p_2) = ((x_{11}, y_{11}), (x_{12}, y_{12}))$ and $s_2 = (q_1, q_2)$ intersect, then intersection(s_1, s_2) is the *N*-point $(\bar{x}, \bar{y}) := (round\_to\_N(x_0), round\_to\_N(y_0))$ where

(i)
$$x_0 = x_{11} + \lambda\,(x_{12} - x_{11})$$
$$y_0 = y_{11} + \lambda\,(y_{12} - y_{11}) \qquad \text{(EQ 2)}$$

x_0 and y_0 are two rational numbers resulting from solving the two equations in exact rational arithmetic (to be implemented through the INT primitives alone). λ is chosen as mentioned in (EQ 1).

(ii) the function *round_to_N* rounds a rational number to the “nearest” number in *N*.

For the function *round_to_N* we give a simple algorithm to show that integer arithmetic is sufficient to calculate the “nearest” number in *N* from a rational number $c = \frac{a}{b}$:

```
function round_to_N (a, b : integer) : integer;
var z : integer;
begin
  if a ≥ b then z := a div b; a := a mod b else z := 0 end; (* now a < b so that 0 < a/b < 1 hold *)
  if a = 0 then return z end;
  if 2 ∗ a ≤ b then return z else return z + 1 end
end round_to_N;
```

Let $s = (p_1, p_2) = ((x_1, y_1), (x_2, y_2))$ be an *N*-segment and let $p = (x, y)$ be an *N*-point. *p* lies on *s*, for short: *p* on *s*, if

(i) $(x_2 - x_1)\,(y - y_1) + (x - x_1)\,(y_1 - y_2) = 0$

(ii) $x \in x\text{-}ext(s) \vee y \in y\text{-}ext(s)$

An *N*-point *p* lies within an *N*-segment *s*, for short: *p* in *s*, if additionally to (i) and (ii) holds

(iii) $x \notin \{x_1, x_2\} \vee y \notin \{y_1, y_2\}$.

One can observe that the largest numbers occur in the equations (EQ 2)

$$x_0 = \frac{x_{11}D + D_1(x_{12} - x_{11})}{D}, y_0 = \frac{y_{11}D + D_1(y_{12} - y_{11})}{D}$$

which leads to the requirement that numbers up to $|2n^3|$ should be representable.

References

[AbB88] Abiteboul, S., and C. Beeri, On the Power of Languages for the Manipulation of Complex Objects. Technical Report 846, INRIA (Paris), 1988.

[BeO79] Bentley, J.L., and T. Ottmann, Algorithms for Reporting and Counting Geometric Intersections. *IEEE Trans. on Computers C-28 (1979)*, 643-647.

[EdM88] Edelsbrunner, H., and E.P. Mücke, Simulation of Simplicity. Proc. ACM Symposium on Computational Geometry (Urbana-Champaign, Illinois), 1988.

[EgFJ89] Egenhofer, M.J., A. Frank, and J.P. Jackson, A Topological Data Model for Spatial Databases. Proc. SSD 89 (Santa Barbara, California), 1989, 271-286.

[GaNT91] Gargano, M., E. Nardelli, and M. Talamo, Abstract Data Types for the Logical Modeling of Complex Data. *Information Systems 16, 5 (1991)*.

[GrY86] Greene, D., and F. Yao, Finite-Resolution Computational Geometry. Proc. 27th IEEE Symp. on Foundations of Computer Science, 1986, 143-152.

[GuiSS89] Guibas, L., D. Salesin, and J. Stolfi, Epsilon-geometry: Building Robust Algorithms from Imprecise Computations. Proc. SIAM Conf. on Geometric Design (Tempe, Arizona), 1989.

[Gut84] Guttman, A., R-Trees: A Dynamic Index Structure for Spatial Searching. Proc. ACM SIGMOD Conf. 1984, 47-57.

[Gü88a] Güting, R.H., Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems. Proc. of the Intl. Conf. on Extending Database Technology (Venice, Italy), 1988, 506-527.

[Gü88b] Güting, R.H., Modeling Non-Standard Database Systems by Many-Sorted Algebras. Fachbereich Informatik, Universität Dortmund, Report 255, 1988.

[Gü89] Güting, R.H., Gral: An Extensible Relational Database System for Geometric Applications. Proc. of the 15th Intl. Conf. on Very Large Databases (Amsterdam, The Netherlands), 1989, 33-44.

[GüS92] Güting, R.H., and M. Schneider, Realm-Based Spatial Data Types. FernUniversität Hagen, Manuscript in preparation, 1992.

[HeSW89] Henrich, A., H.-W. Six, and P. Widmayer, The LSD Tree: Spatial Access to Multidimensional Point- and Non-Point-Objects. Proc. of the 15th Intl. Conf. on Very Large Data Bases (Amsterdam, The Netherlands), 45-53.

[JoC88] Joseph, T., and A. Cardenas, PICQUERY: A High Level Query Language for Pictorial Database Management. *IEEE Trans. on Software Engineering 14 (1988)*, 630-638.

[LiN87] Lipeck, U., and K. Neumann, Modelling and Manipulating Objects in Geoscientific Databases. Proc. 5th Intl. Conf on the Entity-Relationship Approach (Dijon, France, 1986), 1987, 67-86.

[NaME90] Nagy, G., M. Mukherjee, and D.W. Embley, Making Do with Finite Numerical Precision in Spatial Data Structures. Proc. 4th Intl. Symp. on Spatial Data Handling (Zürich, Switzerland), 1990, 55-65.

[NiP82] Nievergelt, J., and F.P. Preparata, Plane-Sweep Algorithms for Intersecting Geometric Figures. *Communications of the ACM 25 (1982)*, 739-747.

[OrM88] Orenstein, J., and F. Manola, PROBE Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Trans. on Software Engineering 14 (1988)*, 611-629.

[OtTU87] Ottmann, T., G. Thiemt, and C. Ullrich, Numerical Stability of Geometric Algorithms. Proc. 3rd ACM Symp. on Computational Geometry, 1987, 119-125.

[RoFS88] Rossopoulos, N., C. Faloutsos, and T. Sellis, An Efficient Pictorial Database System for PSQL. *IEEE Trans. on Software Engineering 14 (1988)*, 639-650.

[ScV89] Scholl, M., and A. Voisard, Thematic Map Modeling. Proc. SSD 89, (Santa Barbara, California), 1989, 167-190.

[SvH91] Svensson, P., and Z. Huang, Geo-SAL: A Query Language for Spatial Data Analysis. Proc. SSD 91 (Zürich, Switzerland), 1991, 119-140.

[Vo92] Voisard, A., Bases de données géographiques: du modèle de données à l'interface utilisateur. Ph.D. Thesis, University of Paris-Sud (Centre d'Orsay), 1992.