Data Structures and Intersection Algorithms for 3D Spatial Data Types

Tao Chen and Markus Schneider*
Department of Computer & Information Science & Engineering
University of Florida
Gainesville, FL 32611, USA
{tachen, mschneid}@cise.ufl.edu

ABSTRACT

Apart from visualization tasks, three-dimensional (3D) data management features are not or only hardly available in current spatial database systems and Geographic Information Systems (GIS). But the increasing demands from application domains like urban planning, geoscience, and soil engineering call for systems that are capable of storing, retrieving, querying, and manipulating the underlying 3D spatial data. Current 3D data models are tailored to specific applications and simple 3D spatial objects only, and available 3D data structures are restricted to main memory representations; thus they lack the ability of handling general and complex 3D spatial objects in a database context. Available algorithms, especially intersection algorithms for 3D objects, usually require special properties like convexity or monotonicity. Therefore, universal intersection algorithms that are capable of handling general 3D spatial objects are currently unknown. This paper proposes a paradigm called *slice representation* as a general data representation method for complex 3D spatial data types. In particular, data structures are developed applying the paradigm to *point3D*, *line3D*, *surface*, and *volume* data types. Two intersection algorithms that involve one argument of type *point3D* are introduced and show the benefit of the slice representation.

Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Spatial databases and GIS

General Terms

Design, Implementation

Keywords

3D data structure, slice representation, intersection

*This work was partially supported by the National Science Foundation under grant numbers NSF-CAREER-IIS-0347574 and NSF-IIS-0915914.

ACMGIS '09 November 4-6, 2009. Seattle, WA, USA

1. INTRODUCTION

Due to the development of new technologies like sensors and laser scanners that yield three-dimensional (3D) data, the need for handling these data is rapidly increasing, and a trend emerges to integrate 3D information into spatial database systems and Geographic Information Systems (GIS). Major progress in 3D GIS has been made on 3D data visualization. However, 3D data management and analysis such as querying, manipulation, 3D map overlay, 3D buffering, and 3D shortest route have been largely neglected in spatial database systems and Geographic Information Systems (GIS). But application domains such as urban planning, soil engineering, aviation, transportation and land use planning, earth science, to name just a few, have shown more and more interest in handling large data sets (e.g., 3D maps) in 3D space from a data management perspective. For example, a spatial query like "Find all the building parts in New Orleans that are below the sea level and therefore might be flooded" might be asked to help the city governor to prepare for a potential disaster. A spatial aggregate query such as "Compute the area of the surface of the new genetics building" can help the construction manager estimate the cost of painting the entire building.

Current 3D data representations like the Tetrahedral Network (TEN), the Constructive Solid Geometry (CSG) tree, and the Octree are quite suitable for visualization but rather inefficient for computation. They all suffer from at least one of the following five problems: First, available data structures for 3D spatial objects are mostly main memory representations and make use of main memory pointers. Therefore, they are not applicable in a database context requiring compact storage structures that can be efficiently transferred as blocks between main memory and disk. An expensive serialization would first be needed to store these objects. Second, some data structures are only able to represent *simple* 3D objects but lack the ability to represent *complex* 3D objects like 3D volumes with cavities and multiple components. Further, in order to enable efficient algorithms, some data structures feature the property of convexity or monotonicity and are thus too restrictive. However, it is important to understand that complex 3D spatial objects without too restrictive constraints are needed in spatial applications, and hence in spatial databases and GIS, and that they are not simply extensions of simple objects from an implementation point of view but require much more effort regarding their data structure design. Third, some data structures like 3D grid, Octtree, and BSP tree decompose 3D objects into

smaller components and thus require large amounts of storage. Fourth, for some 3D data structures like the Doubly-Connected Edge List (DCEL), intersection algorithms have not been documented in the literature. Our own considerations have led to the conclusion that these data representations are quite unsuitable for computing intersections since reorganizations of these data structures into more appropriate data structures would be first needed to have a chance to compute intersections. Fifth, those intersection algorithms that have been designed for 3D spatial objects require very tailor-made internal data representations that are not suitable to implement other 3D operations. Further, they only deal with simple spatial objects and often require convex or monotonic objects. We are interested in a general-purpose data structure for each single complex 3D spatial data type that is universally applicable to a large range of 3D operations so that expensive data structure conversions can be avoided. There are some other problems of available data representations that we do not cover in this paper but that our approach can solve. One of them is the difficulty to check the validity and consistency of 3D objects. For example, the boundary of a 3D volume must be closed and there should be no tangling faces attached. Further, retrieval and update operations on parts of a 3D object as well as operations for overlay and buffering are difficult to implement on some data representations. In summary, data representations that are able to offer support for storing, querying, manipulating, and analyzing 3D spatial data are highly required.

There has been a large consensus in the GIS field that spatial database systems should be used to support GIS applications, store both thematic data and geometric data, and provide basic spatial data management operations. Therefore, spatial database systems should be extended to support *3D spatial data types*. Our so-called *abstract model* [12] presents a careful abstract design of a system of complex 3D spatial data types and the semantics of related operations. However, the emphasis is on completeness, closure, consistency and genericity of the type definitions and the semantics of operations. Thus, it is a high level specification of 3D data types that does not consider effective data structures for the types and efficient algorithms for the operations. An implementation of both 3D data types and 3D operations is not or only hardly available in current spatial database systems and GIS. Among all the operations, the classical *intersection* is a fundamental operation that computes the common part of two spatial objects. It is of interest in many applications and needed, for example, for 3D map overlay in GIS. More importantly, it serves as the basis of a large set of other spatial operations such as topological predicates, set operations, and directional predicates. Although *geometric intersection problems* have had a long tradition as fundamental problems in computational geometry, they are rare for the third dimension due to their complexity, and those that have been developed are not suitable in a spatial database context.

In this paper, we introduce a paradigm called *slice representation* as a general data representation method for 3D spatial data types. *Slice data structures* are used as the underlying representation form for the spatial data types *point3D*, *line3D*, *surface* and *volume* as they have been proposed in [12]. This paper also presents intersection algorithms between a *point3D* object and a *surface* object, and between a *point3D* object and a *volume* object; hence, both algorithms involve an argument of type *point3D*. The reason is that algorithms involving *point3D* objects are the simpler ones in the intersection algorithms family but are complex and sufficient enough to demonstrate the benefits of the slice representation. Moreover, they are, of course, important in applications. For example, in geoscience applications, the different interior layers of the earth can be stored in a table as volumes, and different sites indicating the distribution of chemical elements can be stored as 3D points. A query such as "Find the sites of element *Al* that belongs to the earth mantle layer" can be solved by means of an intersection operation between a *point3D* object and a *volume*. Another example of an intersection query is that walls of a building can be stored in a table as surfaces together with some thematic information such as texture and color, and surveillance cameras can be stored as points in the table indicating their locations. A query like "Find all cameras on the wall with id=7" would be interesting for a building manager to focus on the images sent from cameras in a specific area. Such a query can be solved by calling an intersection operation between a *point3D* object and a *surface* object.

The paper is organized as follows: Section 2 presents related work. Formal definitions of the data structures for all spatial data types are introduced in Section 3. Section 4 describes the algorithms for the selected intersection operations. Finally, Section 5 makes some conclusions and discusses future work.

2. RELATED WORK

3D data representation has had a long tradition in disciplines like Computer Aided Design, Computer Graphics, image processing and robotics. In general, there are two approaches to represent 3D objects, namely boundary representations and volume representations. Boundary representations model 3D objects by storing their lower dimensional boundary elements into data structures like Doubly-Connected Edge List (DCEL), Quad-Edge, and Triangular Irregular Network (TIN). This approach fits visualization purposes but is rather inefficient for spatial operations such as intersection. The internal elements of a 3D object, such as edges and faces, are not organized with any kind of geometric ordering such that an expensive reorganization of these data structures would be needed. Further, since they are main memory structures, it requires additional effort to transfer them into compact storage structures.

Volume representations usually decompose 3D objects into data structures like the Octree, the Constructive Solid Geometry (CSG) tree, and the 3D grid. With these data structures, components of an object can be well organized in a tree-like structure. However, the storage required can become very large when storing objects with large volumes. Further, also these representations are main memory structures and do not fulfil the requirement that spatial databases and GIS need external and compact storage structures.

So far, only a limited number of 3D data models has been proposed for spatial databases and GIS; a review can be found in [14]. We confine ourselves here to the *3D Formal Data Structure* (*3D FDS*) [8] and the *Tetrahedral Network* (*TEN*) [7]. A common feature of the two approaches is that they model 3D points, lines, surfaces, and volumes as 3D spatial data types. The difference is that they describe the types in terms of different sets of primitives. The *Formal*

Data Structure (*3D FDS*) [8] provides the four primitives *node*, *arc*, *face*, and *edge* to represent the four types. Implementations of this model are proposed in [9, 15]. The *Tetrahedral Network(TEN)* [7, 1] employs a simplex-oriented approach and also has four primitives named *tetrahedron*, *triangle*, *arc*, and *node*. Each primitive is the simplest form in its dimension. A volume is composed of *tetrahedra*, a surface of *triangles*, a line of *arcs*, and a point of *nodes*. In [5, 11], the TEN model is implemented and integrated into relational databases. These approaches map the models into relational databases by scattering spatial information over several predefined relation tables. This can cause efficiency and expressiveness problems.

Most of the 3D data models only permit single, connected components; volumes may not contain holes. Further, it is unclear how operations like intersection can be implemented within these data models. In [12], formal definitions for all the 3D spatial data types are given. However, the model is a very abstract design and ignores implementation aspects.

Spatial operations are another important component of a spatial database system. Algorithms need to be designed for operations such as intersection and union. The family of intersection algorithms has had a long tradition in computational geometry, but with a clear emphasis on 2D problems. In the 3D space, early research results have shown several efficient algorithms for computing the intersection of two convex polyhedra. For example, in [6] the time complexity of $O(n \log n)$ is achieved for computing the intersection of two convex polyhedra if their surfaces are pre-triangulated. A reduction of the time complexity to linear time is described in [2]. It is achieved by leveraging a hierarchical representation of two polyhedra developed in [3]. In [4], the proposed algorithm allows a relaxation of one of the two objects to a general object. A summary can be found in [10]. However, efficient intersection algorithms only exist when at least one of the two objects has some special features like convexity. When complex spatial objects are considered, we cannot expect the same degree of efficiency of algorithms as in the 2D case. Although, there have been a few attempts to develop 3D spatial operations as in [13], solutions are based on reducing the intersection problem to pairwise intersection tests between 3D lines and 3D polygons or between two 3D polygons. This is rather inefficient when the two objects contain a large number of polygonal faces. As a result, in spatial database systems and GIS, 3D operations such as intersection and union hardly exist.

3. DATA STRUCTURES FOR 3D SPATIAL DATA TYPES

In this section, we describe in detail the various data structures for representing our 3D spatial data types. Section 3.1 gives an overview of the new paradigm called *slice representation* for representing 3D spatial objects. Section 3.2 introduces *slice units* as the basic construction units of this representation form. Finally, Section 3.3 specifies the slice representations of all 3D spatial data types in terms of sequences of slice units and also provides some basic operators.

3.1 Overview of the slice representation strategy

The definitions of the complex 3D spatial data types at the abstract level [12] focus on closure, consistency, semantics, and generality. These data types express spatial objects as infinite point sets in the 3D space. The consequence is that these objects are not directly representable in a computer requiring finite representations. Therefore, in a so-called *discrete model*, we explore discrete data structures for these objects. For example, a 3D line object can be a smooth curve in the abstract model but the representation in the discrete model is usually a linear approximation as a 3D polyline. More importantly, an appropriate data representation at the discrete level must have an expressiveness that comes as close as possible to the semantics of the corresponding data type at the abstract level. Cases such as volumes with cavities and surfaces that meet at a single point on their boundaries must be handled by the data representation. Further, more issues need to be considered when embedding data structures into databases for querying and analysis. 3D data should be organized in a sequential order to enable fast retrieval and to support efficient algorithms.

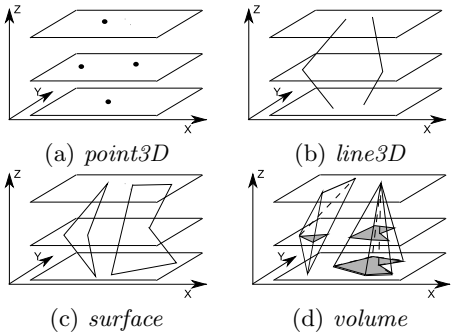


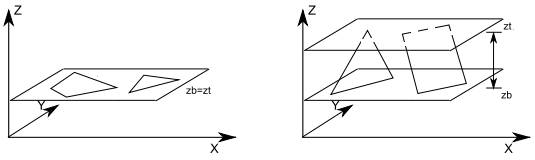
Figure 1: Slice representation of spatial objects

In this paper, we describe a *slice representation* for 3D spatial data types. The basic idea is to decompose a 3D spatial object into a sequence of fragments called *slices*. This *slicing* procedure is done along the z-axis, and the union of all slices then constructs the entire object. Figure 1 shows examples of slice representations for different 3D spatial data types. A major advantage of applying the slice representation paradigm in a database is that, by decomposing a large complex spatial object into a sequence of ordered non-overlapping slices, the quick access of slice units is enabled. As a consequence, in the database context, large objects do not need to be loaded entirely into memory for operations, instead, in most cases only slices that are relevant will be loaded and expensive operations, e.g. intersection, only perform on specific slices.

3.2 Slice units

In this subsection, we introduce new primitives called *slice units* as the basic construction units of our 3D spatial data types. In particular, we specify four kinds of *slice units* named, *spoint*, *sline3D*, *ssurface*, and *svolume*, i.e., one for each spatial data type.

Since our slice representation is based on the linear approximation of 3D objects, e.g. line by segments, surface by polygonal faces, every 3D object contains so called *vertices*. The vertices of a line object are the end points of its segments, and the vertices of a surface object are the joint points of its boundary segments. Similarly, the vertices of a volume object are the joint points of its polygonal faces. The idea of slice representation is to decompose a spatial object



(a) an example of a thin slice

(b) an example of a thick slice

Figure 2: Two types of slices

into a sequence of non-overlapping pieces, namely *slices*, by cutting the object on its n vertices with n' planes ($n \geq n'$) $P_1, P_2, ..., P_{n'}$ that are perpendicular to the z-axis. Each of these planes contains at least one vertex from the object, and the union of n' planes contains all n verticies from the object. In general, a slice is a piece of the object that lies either in between two adjacent cutting planes P_i and P_{i+1}, or on the cutting plane P_i. Therefore, each slice can be attached with a unique *z-interval* $I_z = \{(z_b, z_t) \,|\, z_b, z_t \in Real, z_b \leq z_t\}$, that identifies the location of that slice in the object on z direction. A slice is called a *thin slice* if it has an empty z-interval, i.e. $z_b = z_t$; otherwise, it is called a *thick slice*. An object does not have an extent on z-dimension if all points in the object have the same z coordinate. For example, a polygon perpendicular to z-axis does not have extent on z-dimension. Therefore, a thin slice contains only the piece of a spatial object that does not have extent on z-dimension. To distinguish with a thin slice, we define a thick slice as a slice that contains only the piece of a spatial object that has extent on z-dimension. Figure 2 shows examples of the two types of slices.

A *simple point* in 3D space is represented by a triplet with three coordinates. Let *Point* denote the set of all simple points in 3D space, then we have $Point = \{(x, y, z) \,|\, x, y, z \in Real\}$. A value of the spatial data type *point3D* is a finite set of isolated points in 3D space, and we call an object of this type *complex point*. Since a simple point in 3D space does not have extent, a slice of a simple point is the point itself. For a set of simple points, a slice contains a subset of points that share the same z-coordinate. Therefore, a slice of a *point3D* type object is always a thin slice with an empty z-interval (i.e. $I_z.z_b = I_z.z_t$). In the database context, points are ordered so that a binary search is possible. Since a slice of a *point3D* object contains points with the same z-coordinate, we order them in a (x,y)-lexicographic order. So for any two points $p_1, p_2 \in Point$, we have $p_1 < p_2 \Leftrightarrow p_1.x < p_2.x \vee (p_1.x = p_2.x \wedge p_1.y < p_2.y) \vee (p_1.x = p_2.x \wedge p_1.y = p_2.y \wedge p_1.z < p_2.z)$. Then we can give the definition for the slice unit *spoint* as follows:

$$\begin{aligned} spoint = \{&(pos, \langle p_1, \ldots, p_n \rangle, it) \,|\, pos, n \in Integer, \\ &it \in I_z, 0 \leq pos \leq n, n \geq 0 \\ &\forall 1 \leq i < j \leq n : p_i, p_j \in Point,\ p_i < p_j, \\ &p_i.z = p_j.z = it.z_b = it.z_t\} \end{aligned}$$

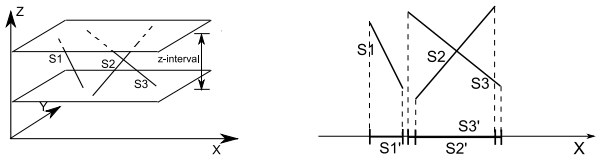
The data structure contains an array of an ordered point sequence $\langle p_1, \ldots, p_n \rangle$ together with a pointer *pos* indicating the current position within the sequence, and a z-interval *it* indicating the location of the slice.

A *line3D* object is formally defined in the abstract model as the union of the images of a finite number of continuous mappings from 1D space to 3D space. A value of this type is called a *complex line*. At the discrete level, a complex line is approximated with a collection of segments in the 3D space. A slice of a line3D object consists of a set of segments. Let *Segment* denote the set of all segments in 3D space, then we have $Segment = \{(p, q) \,|\, p, q \in Point,\ p < q\}$. The equality of two segments $s_1 = (p_1, q_1)$ and $s_2 = (p_2, q_2)$ is defined as $s_1 = s_2 \Leftrightarrow (p_1 = p_2 \wedge q_1 = q_2)$. If the two end points of a segment have same z coordinates, then the segment does not have extent on z dimension. Thin slices of a line3D object contain only segments without extent on z dimension, while thick slices of a line3D object contain segments that have extent within a non-empty z-interval. A thick slice s_i with the z-interval (z_i, z_{i+1}) is bounded by two cutting planes k, given by equation $z = z_i$, and l, given by equation $z = z_{i+1}$. Thus, the two end points of any segment in s_i are contained in the two planes k and l.

Further, we organize the segments in a slice in a certain order, so that a fast retrieval of a segment in a slice can be possible. We project the 3D segments on the x-axis, and align the intervals on the x-axis. However, the intersection free segments in the 3D space can yield overlapping projection intervals on the x-axis. Therefore, in order to order the intervals, we need to capture both the beginning of an interval and the end of an interval. So we store *half segments* for line slices. A half segment is a segment with a dominating point. Let *HSegment* denote the set of half segments in 3D space. We define $HSegment = \{(s, d) \,|\, s \in Segment,\ d \in \{left, right\}\}$. The dominating point of a half segment is indicated by d, if $d = left$ ($right$), then the left (right) end point $p(q)$ of a segment s is the dominating point. Therefore, a segment is represented by two half segments called *twin half segments* with the same end points but different dominating points. We store half segments and order them with respect to their dominating points. For two half segments $h_1 = (s_1, d_1)$ and $h_2 = (s_2, d_2)$, let $isTwin(h_1, h_2)$ be the function that returns true if the two half segments are twin half segments. Further, let dp be the function which yields the dominating point of a half segment, and ndp be the function that returns the end point that is not a dominating point of the half segment. Then we define the order as: $h_1 < h_2 \Leftrightarrow dp(h_1) < dp(h_2) \vee ((dp(h_1) = dp(h_2)) \wedge (ndp(h_1) < ndp(h_2)))$. Let *thin_sline3D* and *thick_sline3D* denote a thin slice unit and a thick slice unit respectively, then we have:

$$\begin{aligned} thin\_sline3D = \{&(pos, \langle hs_1, \ldots, hs_n \rangle, it) \,| \\ &pos, n \in Integer, 0 \leq pos \leq n, n \geq 0 \\ &(i)\ \forall 1 \leq i < j \leq n : \\ &\quad hs_i, hs_j \in HSegment, hs_i < hs_j \\ &(ii)\ \forall 1 \leq i \leq n\ \exists 1 \leq k \leq n : \\ &\quad isTwin(hs_i, hs_k) = true \\ &(iii)\ \forall 1 \leq i \leq n : it \in I_z, \\ &\quad hs_i.p.z = hs_i.q.z = it.z_b = it.z_t\} \end{aligned}$$

$$\begin{aligned} thick\_sline3D = \{&(pos, \langle hs_1, \ldots, hs_n \rangle, it) \,| \\ &pos, n \in Integer, 0 \leq pos \leq n, n \geq 0 \\ &(i)\ \forall 1 \leq i < j \leq n : \\ &\quad hs_i, hs_j \in HSegment, hs_i < hs_j \\ &(ii)\ \forall 1 \leq i \leq n : \exists 1 \leq k \leq n \\ &\quad isTwin(hs_i, hs_k) = true \\ &(iii)\ \forall 1 \leq i \leq n : it \in I_z, \\ &\quad it.z_b < it.z_t, \\ &\quad (hs_i.p.z = it.z_b \wedge hs_i.q.z = it.z_t) \\ &\quad \vee (hs_i.q.z = it.z_b \wedge hs_i.p.z = it.z_t)\} \end{aligned}$$



(a) segments in a slice

(b) projections of segments on the x-axis

$< (s1, left), (s1, right), (s3, left), (s2, left), (s2, right), (s3, right) >$

(c) an ordered sequence of half segments in a slice

Figure 3: An example of a slice of a line3D object

The data structure for both *thin_sline3D* and *thick_sline3D* contains an array of ordered half segments $\langle hs_1, \ldots, hs_n \rangle$ together with a pointer *pos* indicating the current position within the sequence. Condition (i) for both data structures ensures the ordering of the half segments in the sequence. Condition (ii) requires that both twin half segments are contained in a sequence. Condition (iii) for *thin_sline3D* defines an empty z-interval and ensures that no segment in the slice has an extent on the z-direction. Condition (iii) for *thick_sline3D* defines a non-empty z-interval and ensures that the two end points of any segment in the slice are contained inside the two bounding planes determined by the z-interval. A slice unit for a *line3D* object is either a *thin_sline3D* value or a *thick_sline3D* value. We therefore give a uniform definition for the slice unit *sline3D*. Figure 3 gives an example of a slice unit of a *line3D* object.

$$sline3D = \{sl \mid sl \in thin\_sline3D \vee sl \in thick\_sline3D\}$$

As defined in the abstract model, a *surface* is the union of the images of a finite number of continuous mappings from 2D space to 3D space. A value of this type is called *complex surface*. At the discrete level, a complex surface object is approximated with a set of simple 3D polygons possibly with holes. A simple 3D polygon is a simple polygon with co-planner vertices. A slice of a *surface* object consists of a set of simple 3D polygons. Since thin slices and thick slices have different properties, we introduce the definitions for them separately. We first show that in a thick slice, where all simple polygons have extent on the z dimension, a simple polygon does not have holes, and is either a triangle or a trapezoid.

Lemma 1. Let P denote a simple polygon belonging to a thick slice s_i within a z-interval (z_i, z_{i+1}), where $z_i < z_{i+1}$. Then P does not have any holes and is either a triangle or a trapezoid.

Proof. According to the definition of a slice, no vertices of P lies in between the two planes $K : z = z_i$ and $L : z = z_{i+1}$. Further, since P belongs to a thick slice, P does not overlap with K nor L. Thus, for any vertex V of P, V is either on the plane K or on the plane L. If there exist more than three non-collinear vertices of P on K, then the three vertices uniquely determine the plane K. Since all vertices uniquely determine the plane that P lies on, then P overlaps with K, which contradicts with the condition that P belongs to a thick slice. Therefore, K contains only less than three vertices from P. The same can be proved for L. As a result, a total of less than or equal to four vertices are allowed for P, this leads to the conclusion that P does not have any holes, and is either a triangle or a trapezoid. □

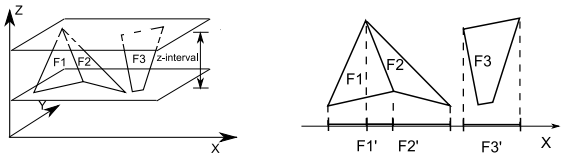
We use *tface* to name a triangle or a trapezoid within a thick slice. A *tface* within a thick slice is defined as:

$$\begin{aligned} tface = \{&(p_1, p_2, p_3, p_4) \mid p_1, p_2, p_3, p_4 \in Point, \\ &p_1, p_2, p_3 \text{ and } p_4 \text{ are coplanar}; \\ &(i)\ p_1.z = p_4.z, p_2.z = p_3.z, p_1.z < p_2.z; \\ &(ii)\ p_1 \le p_4, p_2 \le p_3\} \end{aligned}$$

A *tface* is represented by a list of four coplanar vertices. In the above definition, condition (i) defines p_2 and p_3 to be the two end points that form the upper edge of the *tface*, while p_1 and p_4 is defined to be the two end points of the lower edge of the *tface*. Condition (ii) ensures p_1 and p_2 to be the left end points of the lower edge and the upper edge respectively. Further, if $p_1 = p_4$ or $p_2 = p_3$, then the *tface* is a triangle.

A thick slice of a surface object contains a set of *tfaces*, which are triangles and trapezoids. A constant running time is achieved when computing the intersection of two *tfaces*. Further, we order *tfaces* with respect to their projection intervals on the x-axis, so that a fast retrieval of a specific *tface* is possible. An interval can be obtained by projecting a *tface* on the x-axis, and the two end points of the interval correspond to the leftmost point and the rightmost point of the *tface*. This ordering can help with fast detection of possible intersecting pairs of *tfaces*. For example, two *tfaces* intersect only if their projection intervals on the x-axis overlap. However, to be able to do this, we need to capture both the leftmost point of a *tface* and the rightmost point of a *tface* in the ordered sequence of *tfaces*. So we store *half tfaces* and order them with respect to their *dominating extreme points*. An *extreme point* of a *tface* is either the leftmost point or the rightmost point. A *tface* therefore can be split into two twin half *tfaces*, left half *tface* with the left extreme point to be the dominating extreme point and right half *tface* with the right extreme point to be the dominating extreme point. Let *Htface* denote a half *tface*, then we have $Htface = \{(tf, d) \mid tf \in tface,\ d \in \{left, right\}\}$. In the definition, d is a flag indicating the type of a *tface*. If $d = left$ ($d = right$), then it is a left (right) half *tface* with the left (right) extreme point as dominating extreme point. Let *isTwin* denote the function that returns true if two half *tfaces* are twin. Let *dep* be the function that yields the dominating extreme point of a half *tface*, and *ndep* be the function that returns the non-dominating extreme point. Further, we define the order of two half *tfaces*. Let hf_1, hf_2 be two half *tfaces*, we have $hf_1 < hf_2 \Leftrightarrow dep(hf_1) < dep(hf_2) \vee (dep(hf_1) = dep(hf_2) \wedge ndep(hf_1) < ndep(hf_2))$. With the order of half *tfaces*, we are now ready to define a thick slice unit of a surface object, *thick_ssurface*.

$$\begin{aligned} thick\_ssurface = \{&(pos, \langle hf_1, \ldots, hf_n \rangle, it) \mid \\ &pos, n \in Integer, 0 \le pos \le n, n \ge 0 \\ &(i)\ \forall 1 \le i < j \le n : \\ &\quad hf_i, hf_j \in Htface, hf_i < hf_j \\ &(ii)\ \forall 1 \le i \le n : \exists 1 \le k \le n \\ &\quad isTwin(hf_i, hf_k) = true \\ &(iii)\ \forall 1 \le i \le n : it \in I_z, \\ &\quad hf_i.p_1.z = it.z_b, hf_i.p_2.z = it.z_t\} \end{aligned}$$



(a) *tfaces* in a thick slice (b) projection intervals of *tfaces* on the x-axis

$< (F1, left), (F2, left), (F1, right), (F2, right), (F3, left), (F3, right) >$

(c) an ordered sequence of half *tfaces* in a slice

Figure 4: An example of a thick slice of a surface object

The data structure for *thick_ssurface* unit is an array of ordered half *tfaces* $\langle hf_1, \ldots, hf_n \rangle$ together with a pointer *pos* indicating the current position within the sequence. Condition (i) ensures that all *tfaces* in the thick slice are ordered from left to right. Condition (ii) requires that both twin half *tfaces* are stored in the sequence. Condition (iii) defines a non-empty z-interval and ensures that the vertices of any half *tface* in the slice are contained within the two bounding planes determined by the z-interval. Figure 4 gives an example of a thick slice unit of a surface object.

The other slice type of a surface is the thin slice. A thin slice contains 3D polygons that are coplanar and perpendicular to the z-axis. Any simple 3D polygon in a thin slice can have holes. In fact, a thin slice is a 2D region object with a z elevation. A simple polygon consists of a number of segment cycles. A cycle is formed by segments linked in cyclic order, having always the interior of the polygon at their right side by viewing from the top. Similar to slice unit *sline3D*, we store half segments in a thin slice in the order of their dominating points for fast retrieval purpose. Thus, in order to reflect the cyclic order of segments in a thin slice, we add an extra field $*ns_i$ for any segment hs_i called *next_in_cycle* to indicate the next segment in a cycle. Further, we add two additional arrays to store the cycles and faces. The cycles array $\langle (*fs_1, *nc_1), \ldots, (*fs_m, *nc_m) \rangle$ keeps a record for each cycle $C_i (1 \leq i \leq m)$ in the thin slice, containing a pointer $*fs_i$ to the first half segment in the cycle C_i and a pointer $*nc_i$ to the next cycle within the same polygon. The faces array $\langle *fc_1, \ldots, *fc_r \rangle$ stores one record per polygon, with a pointer $*fc_j (1 \leq j \leq r)$ to the first cycle in the polygon P_j. Let *thin_ssurface* denote the thin slice of a surface object, we have

$$
\begin{aligned}
\mathit{thin\_ssurface} = \{ & (\mathit{pos}, \langle (hs_1, *ns_1), \ldots, (hs_n, *ns_n) \rangle, \\
& \langle (*fs_1, *nc_1), \ldots, (*fs_m, *nc_m) \rangle, \\
& \langle *fc_1, \ldots, *fc_r \rangle, it) \mid \\
& \mathit{pos}, n, m, r \in \mathit{Integer}, 0 \leq \mathit{pos} \leq n, \\
& 0 \leq r < m < n; \\
& (i)\ \forall 1 \leq i < j \leq n : \\
& \quad hs_i, hs_j \in \mathit{HSegment}, hs_i < hs_j \\
& (ii)\ \forall 1 \leq i \leq n : it \in I_z, \\
& \quad hs_i.p.z = hs_i.q.z = it.z_b = it.z_t \\
& (iii)\ \forall 1 \leq i \leq n\ \exists 1 \leq k \leq n : \\
& \quad \mathit{isTwin}(hs_i, hs_k) = \mathit{true} \}
\end{aligned}
$$

The data structure for a thin slice of a surface object consists of three arrays. In the definition, n is the total number of half segments in a thin slice, m is the number of cycles in a thin slice, and r is the number of polygons in a thin slice. Finally, a slice unit $ssurface$ of a surface object is either a thick slice unit *thick_ssurface* or a *thin_ssurface*. So we have:

$$\mathit{ssurface} = \{ss | ss \in \mathit{thin\_ssurface} \vee ss \in \mathit{thick\_ssurface}\}$$

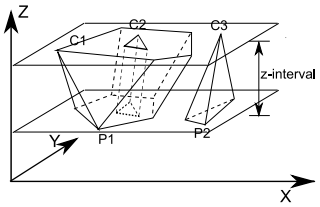


Figure 5: An example of a volume slice

The *volume* data type defined in the abstract model describes complex volumes that may contain disjoint components, and each component may have cavities. Moreover, the *volume* type defined in the abstract model allows non-manifolds, e.g. two volumes meet at a single point. At the discrete level, a volume object consists of one or several subsets of the space enclosed by its boundary, which is approximated with a set of polygonal faces. A slice of a volume object is enclosed by a slice of its surface, which is a collection of *tfaces*. Further, a volume slice can only be a thick slice since it always have extent on the z dimension. A volume slice contains one or several disconnected solids, whereas each solid consists of a number of *tface* cycles. A cycle is formed by tfaces linked in cyclic order, having always the interior of the solid at the right side by viewing from the top. Thus, similar to surface slice unit, we store half *tfaces* in the order of their dominating extreme points. However, in order to reflect the cyclic order of the *tfaces* in a volume slice, we add an extra field $*ns_i$ for any half *tface* hf_i namely *next_in_cycle* to indicate the next half *tface* in a *tface* cycle. Further, we add two additional arrays to store the cycles and solids. The cycles array $\langle (*ff_1, *nc_1), \ldots, (*ff_m, *nc_m) \rangle$ keeps a record for each cycle $TF_i (1 \leq i \leq m)$ in the volume slice, containing a pointer $*ff_i$ to the first half *tface* in the cycle TF_i and a pointer $*nc_i$ to the next cycle within the same solid. The solids array $\langle *fc_1, \ldots, *fc_r \rangle$ stores one record per solid, with a pointer $*fc_j (1 \leq j \leq r)$ to the first cycle in the solid P_j. Let *svolume* denote the slice of a volume object, we have

$$
\begin{aligned}
\mathit{svolume} = \{ & (\mathit{pos}, \langle (hf_1, *nf_1), \ldots, (hf_n, *nf_1) \rangle, \\
& \langle (*ff_1, *nc_1), \ldots, (*ff_m, *nc_m) \rangle, \\
& \langle *fc_1, \ldots, *fc_r \rangle, it) \mid \\
& \mathit{pos}, n, m, r \in \mathit{Integer}, 0 \leq \mathit{pos} \leq n, \\
& 0 \leq r < m < n; \\
& (i)\ \forall 1 \leq i < j \leq n : \\
& \quad hf_i, hf_j \in \mathit{Htface}, hf_i < hf_j \\
& (ii)\ \forall 1 \leq i \leq n : it \in I_z, \\
& \quad it.z_b < it.z_t, hf_i.p_1.z = it.z_b, \\
& \quad hf_i.p_2.z = it.z_t \\
& (iii)\ \forall 1 \leq i \leq n\ \exists 1 \leq k \leq n : \\
& \quad \mathit{isTwin}(hf_i, hf_k) = \mathit{true} \}
\end{aligned}
$$

The data structure for a slice of a volume object consists of three arrays. In the definition, n is the total number of half *tfaces* in a volume slice; m is the number of cycles formed by *tfaces*; r is the number of solids in a volume slice. Figure 5

shows an example of a volume slice. The slice contains 24 half *tfaces*, 3 cycles (C_1, C_2, C_3) and 2 polyhedra (P_1, P_2). P_1 consists of two cycles C_1 and C_2, and P_2 consists of one cycle C_3.

3.3 Slice representation of 3D spatial data types and basic operators

A spatial object of any type consists of a sequence of slice units, where slice units are ordered according to the z-intervals. Thus, we first give the definition for the ordering of two z-intervals. Let $I_1 = \{z_{b1}, z_{t1}\}$, $I_2 = \{z_{b2}, z_{t2}\}$, then we have $I_1 < I_2 \Leftrightarrow z_{t1} \leq z_{b2}$. The equality of two z-intervals are defined as $I_1 = I_2 \Leftrightarrow (z_{b1} = z_{b2}) \wedge (z_{t1} = z_{t2})$. Then we can define the four spatial data types *point3D*, *line3D*, *surface*, and *volume* as following:

$$\begin{aligned} point3D = \{(pos, \langle ps_1, \ldots, ps_n\rangle) \mid \\ pos, n \in Integer, 0 \leq pos \leq n, n \geq 0, \\ \forall 1 \leq i < j \leq n: \ ps_i, ps_j \in spoint, \\ ps_i.it < ps_j.it\} \end{aligned}$$

$$\begin{aligned} line3D = \{(pos, \langle ls_1, \ldots, ls_n\rangle) \mid \\ pos, n \in Integer, 0 \leq pos \leq n, n \geq 0, \\ \forall 1 \leq i < j \leq n: \ ls_i, ls_j \in sline3D, \\ ls_i.it < ls_j.it\} \end{aligned}$$

$$\begin{aligned} surface = \{(pos, \langle ss_1, \ldots, ss_n\rangle) \mid \\ pos, n \in Integer, 0 \leq pos \leq n, n \geq 0, \\ \forall 1 \leq i < j \leq n: \ ss_i, ss_j \in ssurface, \\ ss_i.it < ss_j.it\} \end{aligned}$$

$$\begin{aligned} volume = \{(pos, \langle vs_1, \ldots, vs_n\rangle) \mid \\ pos, n \in Integer, 0 \leq pos \leq n, n \geq 0, \\ \forall 1 \leq i < j \leq n: \ vs_i, vs_j \in svolume, \\ vs_i.it < vs_j.it\} \end{aligned}$$

The data structure for any spatial data types consists of an array of ordered slices together with a pointer *pos* indicating the current position within the sequence. In order to manipulate the data structures, several operations can be provided for retrieving information from the data structures. Due to the space limitation, we provide only a few basic operations which will be used by the algorithms in the following sections. For all the operations, we first give the syntax, then we describe the semantics of the operations.

$$\begin{aligned} &\langle \alpha_1, \alpha_2, \alpha_3, \alpha_4\rangle = \langle spoint, thin\_ssurface, thick\_ssurface, \\ &\qquad svolume\rangle \\ &\langle \beta_1, \beta_2, \beta_3, \beta_4\rangle = \langle point, HSegment, Htface, Htface\rangle \\ &\langle \gamma_1, \gamma_2, \gamma_3\rangle = \langle point3D, surface, volume\rangle \\ &\langle \lambda_1, \lambda_2, \lambda_3\rangle = \langle spoint, ssurface, svolume\rangle \\ &\forall i \in \{1, 2, 3, 4, 5, 6\} \end{aligned}$$

$$\begin{aligned} &get\_first && : \alpha_i && \rightarrow \beta_i \\ &get\_next && : \alpha_i && \rightarrow \beta_i \\ &end\_of\_array && : \alpha_i && \rightarrow bool \\ &proj\_x && : \beta_1 && \rightarrow (Real) \\ &&& : \beta_i && \rightarrow (Real, Real)\ (i \neq 1) \end{aligned}$$

$$\begin{aligned} &get\_first\_slice && : \gamma_i && \rightarrow \lambda_i \\ &get\_next\_slice && : \gamma_i && \rightarrow \lambda_i \\ &end\_of\_seq && : \gamma_i && \rightarrow bool \\ &is\_empty && : \gamma_i && \rightarrow bool \\ &interval\_z && : \gamma_i && \rightarrow I_z \end{aligned}$$

The *get_first* operation returns the first element in the data array of a slice primitive and set the *pos* pointer to 1. The *get_next* operation returns the next element of current position in the array, and increments the *pos* pointer. The predicate *end_of_array* yields *true* if $pos = n$. The operation *proj_x* projects the operand on the x-axis, and returns an interval on the x-axis.

The *get_first_slice* operation returns the first slice in the slice sequence of a spatial object and set the *pos* pointer to 1. The *get_next_slice* operation returns the next slice of current position in the sequence, and increments the *pos* pointer. The predicate *end_of_seq* yields *true* if $pos = n$, and the predicate *is_empty* yields *true* if $n = 0$. The operation *interval_z* returns the z-interval of the operand slice. All above operations have constant cost $O(1)$.

Further, since our intersection algorithms that involve at least one operand as a *point3D* object yields a new *point3D* object, we provide a few functions for creating a new *point3D* object. The *new_point3D* function creates a new empty *point3D* object with an empty slice sequence where $n = 0, pos = 0$. The *new_point_slice* function creates a new empty point slice with an empty point array where $n = 0, pos = 0, I_z = (0, 0)$. The other two operations, *insert_point* and *insert_point_slice*, insert a point and a point slice into data array respectively, and both increase *pos* and n by 1.

4. SELECTED INTERSECTION ALGORITHMS FOR 3D SPATIAL OPERATIONS

Spatial operations are another important component in a spatial database system. They are integrated into database systems and used as tools for manipulating spatial data. Due to the space limitation, we only focus on the geometry set operation *intersection*. According to the signature of *intersection* operation in the abstract model, an intersection operation involves two operands and produces a spatial object of a dimension lower or equal to the lower-dimensional operand. For any combination of the operands, a corresponding algorithm needs to be designed. In this paper, we introduce two algorithms for the intersection operation between a point3D object and a surface, and the intersection operation between a point3D object and a volume.

$$\begin{aligned} &intersection: \ surface \times point3D \rightarrow point3D \\ &intersection: \ volume \times point3D \rightarrow point3D \end{aligned}$$

For both operand combination, we introduce two separate algorithms *sp3D_intersect* and *vp3D_intersect*.

The algorithm *sp3D_intersect* computes the intersection between a *surface* object and a *point3D* object. The first step of the algorithm is to perform a parallel scan on the two ordered slice sequences from both objects so that possible intersecting slice units can be picked for testing. This leads to the problem how to compute the intersection between a surface slice unit and a point slice unit. We treat the two types of surface slices, the thin surface slice *thin_ssurface* and the thick surface slice *thick_ssurface*, separately. A thick surface slice contains *half tfaces*, which are either triangles or trapezoids. All half *tfaces* are ordered from left to right with respect to their dominating extreme points. Therefore, for any two slices, a surface slice *ss* and a point slice *sp*, we apply an algorithm using the *plane sweep* paradigm to find out all possible *tface-point* intersecting pairs within the two

```
(1)  method sp3D_sweep (ss, sp)
(1)    sc ← new_point_slice, SL ← new empty hash
(2)    p ← get_first(sp), hf ← get_first(ss)
(3)      while !end_of_array(sp)
(4)            and !end_of_array(ss) do
(5)          if p ≥ dep(hf) then
(6)            if hf.d ==left then
(7)              insert hf into the hash array SL
(8)            else
(9)              remove the twin of hf
(10)             from the hash array SL
(11)           endif
(12)           hf ← get_next(ss)
(13)         else
(14)           for i = 0 to the number of
(15)             elements in sweep status list
(16)             if point_on_tface(p, SL[i]) then
(17)               sc ← insert_point(p, sc)
(18)             endif
(19)           endfor
(20)         endif
(21)     endwhile
(22)   return sc
       end
```

Figure 6: The sweeping algorithm *sp3D_sweep* for a thick surface slice *ss* and a point slice *sp*

slices *ss* and *sp*. In the algorithm, we maintain two lists, a static *event points list* and a dynamic *sweep status list*(SL). A vertical plane parallel to the z-axis sweeps the space from left to right at special points called *event points* stored in the event points list. In our case, the dominating extreme points of the half *tfaces* in the surface slice are merged with the points in the point slice, and are stored as event points in the event points list. Since points in the point slice are ordered and the half *tfaces* in the surface slice are ordered according to their dominating extreme points, this merge step would only take $O(t + r)$ time, where t is the number of half *tfaces* in the surface slice *ss* and r is the number of points in the point slice *sp*. When the sweeping plane reaches a dominating point of a half *tface*, the sweep status list will be updated. If the sweeping plane encounters a left half *tface*, which means that the sweeping plane just starts to intersect a *tface*, then the corresponding half *tface* is inserted into the sweep status list; if the sweeping plane encounters a right half *tface*, then its twin left half *tface* is removed from the sweep status list, meaning that the sweeping plane is leaving that *tface*. As a result, the sweep status list keeps all *tfaces* that are currently intersected by the sweeping plane. Further, when the sweeping plane reaches a point object, all *tfaces* within the sweep status list are tested against the point for intersection. We implement the sweep status list as a hash table, so that the insertion and removal operation can be done with $O(1)$ time in most cases. Further, let the predicate *point_on_tface* be the predicate that performs the intersection test for a *tface* and a point, which returns true if the point is on the *tface*. Since the number of edges of a *tface* is either 3 for a triangle or 4 for a trapezoid, the running time of the predicate *point_on_tface* is constant. Figure 6 describes the sweeping algorithm *sp3D_sweep* for a

```
(1)  method sp3D_sweep′ (ts, sp)
(1)    sc ← new_point_slice, SL ← new empty hash
(2)    p ← get_first(sp), hs ← get_first(ts)
(3)      while !end_of_array(sp)
(4)            and !end_of_array(ts) do
(5)          if p ≥ dp(hs) then
(6)            if hs.d ==left then
(7)              insert hs into the hash array SL
(8)            else
(9)              remove the twin of hs
(10)             from the hash array SL
(11)           endif
(12)           hs ← get_next(ts)
(13)         else
(14)           ray ← new ray shooting from
(15)                 p to +∞ of y-axis
(16)           cnt ← 0
(17)           for i = 0 to the number of elements
(18)              in the sweep status list
(19)             if ray ∩ SL[i] then
(20)               cnt ← cnt + 1
(21)             endif
(22)           endfor
(23)           if cnt is an odd number then
(24)             sc ← insert_point(p, sc)
(25)           endif
(26)         endif
(27)     endwhile
(28)   return sc
       end
```

Figure 7: The modified sweep algorithm *sp3D_sweep′* for a thin slice *ts* and a point slice *sp*.

thick surface slice *ss* and a point slice *sp*.

The above algorithm computes the intersection between a thick surface and a point slice. Figure 10a gives an example of the sweeping algorithm for a thick surface slice and a point slice. The current event point is P and the current sweep status list contains $F2$. An intersection test is performed for the pair (P, $F2$). However, for a thin surface slice, which does not contain *tfaces*, the algorithm for computing intersection with a point slice is different. Let *ts* denote a thin surface slice and *sp* denote a point slice. Since both *ts* and *sp* have empty z-intervals, in order to be picked and tested in the sweeping phase, they must be coplanar, i.e. $ts.it = sp.it$. As a result, the problem becomes a 2D problem, that finds intersection between points and regions on a plane perpendicular to the z-axis. For any point p, if it does not intersect a region object then a ray starting from p shooting at any direction must intersect even number of edges from the region object. Therefore, for any point p_i in the point slice *sp*, we create a ray that is parallel to the y-axis and shoots from p_i to the $+\infty$ of the y-axis. By counting the number of segments in *ts* that intersects the ray, we can determine if *ts* intersects p_i. We modify the plane sweep algorithm *sp3D_sweep* and name the new algorithm *sp3D_sweep′*, which takes a thin slice *ts* and a point slice *sp* as operands. Figure 7 gives the algorithm.

So far, we have handled both a thin surface slice and a thick surface slice together with a point slice. Now we give

```
(1)  method sp3D_intersect (s, p)
(2)    c ← new_point3D
(3)    if !is_empty(p) and !is_empty(s)
(4)    then sp ← get_first_slice(p)
(5)      ss ← get_first_slice(s)
(6)      while !end_of_seq(sp)
(7)            and !end_of_seq(ss) do
(8)        if interval_z(sp) > interval_z(ss) then
(9)          ss ← get_next_slice(s)
(10)       else if interval_z(sp) < interval_z(ss)
(11)       then sp ← get_next_slice(p)
(12)       else if interval_z(sp) == interval_z(ss)
(13)       then      *If ss is a thin slice
(14)                 and coplanar with sp*
(15)         sc ← sp3D_sweep(ss, sp)′
(16)         c ← insert_point_slice(c, sc)
(17)       else *The z-interval of
(18)         ss contains the z-interval of sp*
(19)         sc ← sp3D_sweep(ss, sp)
(20)         c ← insert_point_slice(c, sc)
(21)       endif
(22)     endwhile
(23)   endif
(24)   return c
(25) end
```

Figure 8: The algorithm *sp3D_intersect* for a surface object s and a point3D object p.

```
(1)  method vp3D_sweep (sv, sp)
(2)    sc ← new_point_slice, SL ← new empty hash
(3)    p ← get_first(sp), hf ← get_first(sv)
(4)      while !end_of_array(sp)
(5)            and !end_of_array(sv) do
(6)        if p ≥ dep(hf) then
(7)          if hf.d ==left then
(8)            insert hf into the hash array SL
(9)          else
(10)           remove the twin of hf from
(11)           the hash array SL
(12)         endif
(13)         hf ← get_next(sv)
(14)       else
(15)         ray ← new ray shooting
(16)           from p to +∞ of the y-axis
(17)         cnt ← 0
(18)         for i = 0 to the number of
(19)           elements in sweep status list
(20)           if ray ∩ SL[i] then
(21)             cnt ← cnt + 1
(22)           endif
(23)         endfor
(24)         if cnt is an odd number then
(25)           sc ← insert_point(p, sc)
(26)         endif
(27)       endif
(28)     endwhile
(29)   return sc
(30) end
```

Figure 9: The sweeping algorithm *vp3D_sweep* for the volume slice sv and the point slice sp.

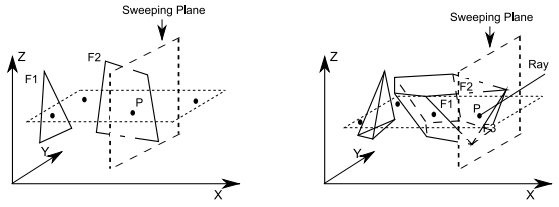
the algorithm *sp3D_intersect* for a surface object s and a point3D object p in Figure 8.

We now analyze the running time of the *sp3D_intersect* algorithm. Let p and s denote a point3D object with n points and a surface object with m segments representing its edges, respectively, where p consists of u slices with r points each slice and where s consists of v slices with either t *tfaces* for each thick slice or t half segments for each thin slice. Then we have $n = ur$ and $m = 2vt$. The plane sweep algorithm for two slices scans the data arrays of both slices once and for all points encountered. It costs $O(K)$ intersection tests where K is the sum of *tface-point* pairs (or *segment-point* pairs) that have their projections on the x-axis intersect. The running time for the *sp3D_intersect* algorithm is $u + v + min(v, u)(r + t + K) \leq u + v + ur + vt + vK < 2m + 2n + vK = O(m + n + vK))$ where vK is much less than m in most cases.

The algorithm *vp3D_intersect* computes the intersection between a *volume* object and a *point3D* object. The first step of the algorithm is the same as the previous algorithms, which involves a parallel scan on the two ordered slice sequences from both objects. Then we develop an algorithm for computing the intersection between a volume slice and a point slice that have overlapping z-intervals. First, we consider the case of testing whether a point is outside of a volume object. If a point is outside of a volume, then the point does not intersect the volume. In other words, if the point is not outside of the volume, then it intersects the volume. Thus, for a point p and a volume v, we create a ray shooting from p to any random direction, and we count the number *cnt* of the polygonal faces of the volume that are intersected by the ray. If the number *cnt* is an odd number, then p intersects volume v. We apply this method when computing intersections between a point slice *sp* and a volume slice *sv*. We also perform a sweeping on the elements of both slices. When a point p from *sp* is encountered, we create a ray that is parallel to the y-axis and shoots from p to the $+\infty$ of the y-axis. Then for all current *tfaces* of the volume slice *sv* in the sweep status list, we test the intersection between them and the ray; meanwhile, we keep a count *cnt* of the intersecting pairs. The point p intersects the volume slice *sv* if *cnt* is an odd number. Further, the intersection test between a *tface* and a ray is trivial and takes only constant time. We give the sweeping algorithm *vp3D_sweep* for the volume slice *sv* and the point slice *sp* in Figure 9.

The above algorithm computes the intersection between a volume slice and a point slice. Figure 10b gives an example of the sweeping algorithm for a volume slice and a point slice. The current event point is P and the current sweep status list contains $F1$, $F2$, and $F3$. All *tfaces* in the sweeping status list are tested against the ray shooting from P. Since P only intersects $F2$, and therefore intersects an odd number of *tfaces*, P intersects the volume slice. Finally, Figure 11 gives the *vp3D_intersect* algorithm for a volume object v and a point object p.

A comparison shows that the time complexity for the intersection algorithm *vp3D_intersect* is the same as for the intersection algorithm *sp3D_intersect*. Thus we need not give a detailed analysis here.



(a) a surface-point slice pair (b) a volume-point slice pair

Figure 10: Sweeping algorithms on slices

```
     method vp3D_intersect (v, p)
(1)    c ← new_point3D
(2)    if !is_empty(p) and !is_empty(v)  then
(3)      sp ← get_first_slice(p)
(4)      sv ← get_first_slice(v)
(5)      while !end_of_seq(sp)
(6)            and !end_of_seq(sv) do
(7)        if interval_z(sp) > interval_z(sv) then
(8)          sv ← get_next_slice(v)
(9)        else if interval_z(sp) < interval_z(sv)
(10)         then sp ← get_next_slice(p)
(11)       else *The z-interval of sv contains
(12)            the z-interval of sp*
(13)         sc ← vp3D_sweep(sv, sp)
(14)         c ← insert_point_slice(c, sc)
(15)       endif
(16)     endwhile
(17)   endif
(18)   return c
     end
```

Figure 11: The *vp3D_intersect* algorithm for a volume object v and a point object p.

5. CONCLUSION AND FUTURE WORK

In this paper, we propose a slice representation as the paradigm of designing data structures for 3D spatial data types. We have given data structures for the four 3D spatial data types *point3D*, *line3D*, *surface*, and *volume* defined at the abstract level, together with two efficient intersection algorithms that rely on the underlying slice representation. The slice representation provides a clean and uniform data structure design and serves as the basis of efficient algorithms. This paper only provides a small collection of operations. Hence, in the future we will supplement it with other important operations which, for example, yield numerical values, or are predicates. Further, constraints for the topology between slices and the topology of elements in the slices can be explored to restrict the slice representations to represent only valid objects, where validity can be defined by different models. For example, some model may only allow manifolds while others allow also non-manifolds. As a consecutive step, validation procedures can be developed to check the validity of inputs.

6. REFERENCES

[1] P. Abdul-Rahman, Alias and Morakot. *Spatial Data Modelling for 3D GIS*. Springer, 2007.

[2] C. Bernard. An optimal algorithm for intersecting three-dimensional convex polyhedra. *SIAM J. Comput.*, 21(4):671–696, 1992.

[3] D. Dobkin and D. Kirkpatrick. Fast detection of polyhedral intersections. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 154–165, 1982.

[4] K. Dobrindt, K. Mehlhorn, and M. Yvinec. A Complete and Efficient Algorithm for the Intersection of a General and a Convex Polyhedron. In *Workshop on Algorithms and Data Structures*, pages 314–324, 1993.

[5] P. v. O. F. Penninga and B. M. Kazar. A Tetrahedronized Irregular Network Based DBMS Approach for 3D Topographic Data Modeling. In *International Symp. on Spatial Data Handling*, pages 581–598, 2006.

[6] S. Hertel, M. Mäntylä, K. Mehlhorn, and J. Nievergelt. Space Sweep Solves Intersection of Convex Polyhedra. *Acta Informatica*, 21(5):501–519, 1984.

[7] P. M. *Integrated modelling for 3D GIS*. PhD thesis, C.T. de Wit Graduate School Production Ecology, the Netherlands, 1996.

[8] M. Molenaar. A Formal Data Structure for The Three Dimensional Vector Maps. In *International Symp. on Spatial Data Handling*, pages 830–843, 1990.

[9] M. Molenaar. A Query Oriented Implementation of A Topologic Data Structure for 3D Vector Maps. In *International Journal of Geographical Information Systems*, pages 243–260, 1994.

[10] D. Mount. *Geometric Intersection*, chapter Handbook of discrete and computational geometry, pages 615–630. CRC Press, Inc., 1997.

[11] F. Penninga and P. van Oosterom. A Compact Topological DBMS Data Structure For 3D Topography. In *Lecture Notes in Geoinformation and Cartography*, pages 455–471, 2007.

[12] M. Schneider and B. E. Weinrich. An Abstract Model of Three-Dimensional Spatial Data Types. In *12th ACM Symp. on Geographic Information Systems (ACM GIS)*, pages 67–72, 2004.

[13] C. Tet-Khuan, A. Abdul-Rahman, and S. Zlatanova. 3D Spatial Operations in Geo DBMS Environment for 3D GIS. In *International conference on Computational Science and Its Applications*, pages 151–163, 2007.

[14] S. Zlatanova, A. A. Rahman, and W. Shi. Topological Models and Frameworks for 3D Spatial Objects. *Journal of Computers & Geosciences*, 30:419–428, 2004.

[15] S. Zlatanova and K. Tempfli. Data Structuring and Visualization of 3D Urban Data. In *Int. Conf. of the Association of Geographic Laboratories in Europe*, 1998.