DESIGN, IMPLEMENTATION, AND INTEGRATION OF DATA TYPES FOR THREE-DIMENSIONAL SPATIAL DATA IN DATABASES

By

TAO CHEN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2012

ACKNOWLEDGMENTS

First, I would like to thank my supervisor, Dr. Markus Schneider, for providing the guidance and support throughout my research. I appreciate his knowledge and skill and his assistance in writing reports. Second, I would like to thank the other members of my committee (Dr. Alin Dobra, Dr. Tamer Kahveci, Dr. Randy Chow and Dr. Esther Adhiambo Obonyo), for the assistance they provided. I would also like to thank my family for the support they provide me through my entire life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

DESIGN, IMPLEMENTATION, AND INTEGRATION OF DATA TYPES FOR THREE-DIMENSIONAL SPATIAL DATA IN DATABASES

By

Tao Chen

August 2012

Chair: Markus Schneider
Major: Computer Engineering

Spatial database systems so far have dealt mostly with two-dimensional spatial entities, and the support for three-dimensional data management has been largely neglected. The primary goal of the proposed research is to develop a new formal data model and type system called Spatial Algebra 3D (*SPAL3D*) for three-dimensional (3D) spatial data in database systems. In this research, we explore the fundamental properties and the structure of complex 3D spatial data from a data management and database perspective, identifies and designs algorithms for their most important operations and predicates, and supports their representation and storage in database systems.

Recently, application domains such as urban planning, soil engineering, aviation, transportation and earth science, to name just a few, have shown more and more interest in handling large data sets (e.g., 3D maps) in the 3D space from a data management perspective. These application areas require not only 3D geometric visualization and spatial analysis methods but also database support for storing, retrieving, and querying the underlying large volumes of 3D data. Consequently, in addition to 2D (planar) and 2.5D (relief, terrain) data, spatial database systems and geographic information systems (GIS) should also incorporate the treatment of 3D data throughout their architecture. However, to our knowledge, only a few 3D data models have been proposed to handle 3D data in spatial database systems, and they are

tailored to specific applications and simple 3D spatial objects only. More importantly, 3D functionalities hardly exist in spatial databases.

The goal of this research is to provide solutions based on three fundamental pillars: (i) an abstract data model (*SPAL3D-A*) for the rigorous, mathematical definition of a comprehensive type system (algebra) for 3D spatial data including volumes, surfaces, and 3D lines, (ii) a discrete data model (*SPAL3D-D*) for the design of effective geometric data structures for the 3D spatial data types of *SPAL3D-A* and on efficient geometric algorithms on these data structures for the 3D operations and predicates of *SPAL3D-A*, and (iii) the implementation and database integration (*SPAL3D-I*) of the data structures and algorithms of *SPAL3D-D* as abstract data types into several extensible DBMS data models and query languages.

CHAPTER 1
INTRODUCTION

1.1 Background and Motivation

Spatial databases store the topologies or the shapes of spatial objects as attributes in the database. Spatial database management systems provide the ability to represent, query, and manipulate the spatial objects efficiently. For almost a decade, most spatial database systems and geographic information systems have been restricted to the handling of two dimensional data. This restriction is from a data modeling and data management perspective rather than a visualization perspective. However, increasingly the third dimension becomes more and more relevant for application domains like pollution control, water supply and distribution, soil engineering, spatial mining, urban planning and development, archaeology, and aviation, to name only a few. Moreover, with the development of new technologies for collecting three-dimensional (3D) data, large volumes of 3D data become available and the need for handling these data is rapidly increasing. As a result, the integration of 3D information into spatial databases and Geographic Information Systems (GIS) to support ad-hoc queries is desired by many application domains. 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 “ Find the area of the surface of the new genetics building” can help the construction manager estimate the cost of painting the entire building. The development of a 3D database system has two overall goals. The first goal is to develop a 3D type system that represents and stores the 3D data, while the second goal is to design and implement a large amount of operations on the 3D data types.

The first goal is to represent and hold 3D data in a database system. This involves the process of defining data types for 3D data, designing a data structure for each data type, and finding a proper storage in a database context for 3D data. To our

knowledge, only a few 3D data models have been proposed to represent and store 3D data in spatial database systems. Most of the data models are tailored to specific applications and simple 3D spatial objects only. Thus they lack the ability of handling general and complex 3D spatial objects. When it comes to the representation of 3D data in database systems, a popular strategy applied by many 3D data models is to decompose 3D data, e.g. volumes, into smaller elements like nodes, edges, and faces and spread them into predefined relational tables. The advantage is obvious that DBMS features like table joins and indexes can be directly used for accessing spatial elements. But this approach is problematic. From a conceptual perspective, the object view of the geometry of a spatial object gets lost. The validation of a spatial object becomes extremely difficult. From an implementation perspective, in order to develop operations like intersection, the geometries have to be reconstructed and be converted to main-memory representations, which is an expensive process. From a storage perspective, since elements of a same spatial object are distributed into relational tables, they might not be clustered. This leads to the increase of the I/O cost for retrieving spatial objects. Therefore, a more plausible strategy is to take an object oriented approach and store 3D spatial data in database systems as abstract data types. The 3D spatial data types should be first class citizen and are no different from other traditional data types like string, integer, and date in databases. Thus in this research, we intend to design a precise and conceptually clean data type system for general and complex 3D spatial objects, and propose data structures for holding them in a database context following an object oriented approach.

The second goal is to extend the existing database management systems with functionalities for querying and manipulating 3D data. Spatial functionalities exist in spatial databases in the form of operations and predicates. Spatial operations play an important role in spatial databases because they are part of spatial queries and are used for performing complex analysis tasks on spatial data. Predicates are a special

kind of operations that yield boolean results. They are frequently used as selection and join conditions in queries. Unfortunately, 3D operations and 3D predicates hardly exist in current spatial databases. Although there exist a large number of algorithms for 3D geometric problems like intersection in computational geometry field which can be candidates for the implementation of operations, they can not be directly used in a spatial database context for the following reasons. First, most algorithms are main memory algorithms that require in memory data structures for holding spatial objects. However, in a database context, this requirement is usually not satisfied because spatial objects are stored on disk and may not be entirely loaded into main memory due to their large size. Second, efficient geometric algorithms usually deal with spatial objects that feature the property of simplicity, convexity or monotonicity. However, spatial objects stored in databases are unpredicted, thus these properties can not be assumed in a database context. It is important to understand that algorithms that work with complex 3D spatial objects without too restrictive constraints are needed for implementing spatial operations and predicates. Third, geometric algorithms require tailor-made internal data structures and the data structures may be different for different algorithms. However, in a database context, we use a unique representation for any spatial object. As a result, if the implementation of different operations are based on different data structures, then expensive data structure conversions are needed when different operations are executed. Therefore, the implementations of 3D operations and 3D predicates are not simply extensions of existing geometric algorithms but require much more effort. It is the goal of this research to bridge this gap and to propose solutions for selected 3D operations and 3D predicates.

1.2 Role of Different Modeling Levels

To achieve the aforementioned goals for developing a 3D spatial database system, we propose solutions at three main levels of abstraction, which we call the abstract level, the discrete level, and the implementation level. The definitions of the structure of spatial

objects as well as of the semantics of operations can be given at these levels. At each level, different models or concepts can be deployed.

A model at the abstract level in the field of spatial databases enables one to express the permissible shapes of spatial objects. It does not have to express semantics in terms of finite representations and is not impeded by computer-specific properties and constraints. For example, a trajectory is described as a 3D curve at the abstract level, which is a (certain kind of) infinite point set in the 3D space without fixing any finite representation and without considering any computer-specific properties. The main advantage of the abstract level is that it is conceptually clean and simple. It is important to start with an abstract model when developing a spatial database system. Many who have developed spatial database systems, both 2D and 3D, have gone straight to implementing the system without formulating an abstract model. The result of starting with the implementation of a system is analogous to when someone writes a program without performing the steps of analysis and design. The system implemented includes only the things of which the implementers think as they implement and is therefore makeshift. In this research, we start with the design and rigorous definition of a comprehensive abstract data model for 3D data, called Spatial Algebra 3D at the Abstract Level (SPAL3D-A). Our algebra framework facilitates an understanding of the nature and properties of 3D data and provides a set of three-dimensional spatial data types, namely point3D, line3D, surface, and volume, together with the signatures and semantics of a collection of spatial operations (like intersection, centroid). In addition, we explore the topological relationships and cardinal direction relationships between two 3D spatial objects, which can be later integrated in queries as predicates.

A discrete model deals with the design of the finite representations for spatial objects and the design of algorithms for operations and predicates in a database context. For example, a trajectory is described as a 3D polygonal line at the discrete level. The advantage of the discrete level is that it is closer to and hence better supports

the implementation of concepts. In this research, we propose effective geometric data structures for the 3D spatial data types defined at the abstract level, and design efficient geometric algorithms on these data structures for the 3D operations and predicates.

The implementation level is where we make connections to the underlying database system. The data structures designed at the discrete level need to be materialized and stored in databases. For example, a trajectory is described as an ordered set of 3D line segments and are stored in relational tables at the implementation level. Algorithms for operations are implemented and operations are integrated into query languages at this level. In this research, we take an object oriented approach and select the Binary Large Object (*BLOB*) that are available in most commercial databases as the underlying data holder for 3D spatial objects. However, BLOBs suffer from the problems of lacking support for efficient updates, tedious byte level data manipulation, and no structure preservation. Thus we introduce a novel data type called Intelligent Binary Large Object (*iBLOB*) that leverages the traditional BLOB type in databases, preserves the structure of spatial objects, and provides smart query and update capabilities.

1.3 Organization of the Dissertation

This dissertation is organized into 6 chapters. The organization of the chapters follows our strategy of developing 3D spatial database systems at three levels of abstractions. Chapter 2 discusses the related work in developing 3D spatial database systems. Chapter 3 introduces an abstract model for 3D spatial objects and their operations. Chapter 4 introduces the data representations of the 3D spatial data types at a discrete level. Chapter 5 introduces the implementation of the 3D data types based on our novel iBlob. Finally, Chapter 6 makes conclusion and describes the future work.

CHAPTER 2
RELATED WORK

In this chapter we discuss related work as far as it is relevant for data modeling approaches for 3D spatial data. So far, data models for 3D spatial data are hardly available for spatial databases and GIS; these systems exclusively model and manage 2D and 2.5D spatial data. We begin with an overview of the 3D data representation strategies in traditional disciplines in Section 2.1. Section 2.2 presents a cross-section of our extensive literature study of many possible and promising application areas that would prosper from 3D spatial data support in GIS. Due to space considerations, we have not listed the numerous references that document these applications. Section 2.3 summarizes the available modeling concepts for 3D spatial data in spatial databases. Finally, as one of the topics in our research work, two most important qualitative spatial relationships between 3D objects, the topological relationships and the directional relationships, are discussed in Section 2.4.

2.1 3D Data Representations in Traditional Disciplines

Research on 3D spatial data modeling in applications has had a long tradition in disciplines like solid modeling, geometric modeling, computer aided design (CAD), constructive solid geometry (CSG), computer graphics, computer vision, and robotics. In general, there are two approaches to represent 3D objects, namely the boundary representation and the volume representation.

Boundary representation (often abbreviated as *B-rep* or *BREP*) is a method for representing shapes using the limits. The limits of a 3D objects are the lower dimensional boundary elements like vertices, edges, and faces. Faces are bounded with edges and edges are bounded with vertices. Boundary representations model 3D objects not only by storing the lower dimensional boundary elements, but also by maintaining their connectivity informations in a data structure. Famous data structures

for this purpose are the Doubly-Connected Edge List (*DCEL*), the *Quad-Edge*[45], and the *Winged Edge*[8, 38].

The volume representation is a decomposition based approach that decomposes a 3D object into simpler components like cubes, tetrahedra and spheres. Then, components of the object can be well organized in smart data structures like trees. One volume representation that is applied in computer aided design (CAD) systems for constructing man made objects like parts of engines is the Constructive solid geometry representation. Constructive solid geometry is a procedural modeling technique using primitives. These primitives are usually predefined objects of simple shapes like *cuboids*, *cylinders*, *prisms*, *pyramids*, *spheres*, and *cones*. Boolean operations such as *union*, *intersection* and *difference* are used to combine primitives to construct a complex solid. By arranging the boolean operations and primitive objects into a binary search tree [94], which is also called the CSG tree, a complex solid can be constructed in a bottom up fashion. Improvements for constructing such a CSG tree can be found in [16]. Another similar technique is the primitive instancing (*PI*) modeling strategy. With PI, instead of using standard predefined primitives, users can defined their own primitives to build complex solids.

Other volume representations exist like the *Voxel model*, the *Octree*[5, 13], and the *BSP tree*[52, 74]. They are in common based on the space partitioning techniques that decompose space into grids or cells. In these representations, a space partition strategy is first determined, i.e., Octree or BSP tree, then a 3D solid is decomposed and organized in the corresponding partition tree structure.

There is a large variety and divergence of concepts and structures on representing 3D geometric objects. In this research, we leverage a number of these concepts but focus especially on their suitability and transferability to spatial database and GIS applications and adapt them correspondingly.

2.2 Database Applications for 3D Spatial Data

Despite of the various 3D models in other disciplines like those mentioned above, in a spatial database and GIS context however, the support for 3D spatial data handling is rather limited. A simple transfer of 3D concepts from other disciplines is impeded since most of their data structures are inappropriate for GIS purposes (for example, the constructive approach in computer aided design) and/or use pure main memory structures (for example, in computer graphics) whereas spatial databases and GIS require external data and index structures. Moreover, most available 3D data models focus mainly on visualizations and representations of 3D spatial data, thus the data structures are usually not optimized for spatial computations. On the other hand, unlike the other disciplines like CAD whose applications require high quality visualizations, a major task of spatial database systems and GIS is the data management and data manipulation [83], which also holds for 3D spatial data. Therefore, the existing data models do not fully satisfy the needs of GIS applications.

The lacking consideration of 3D data management in spatial database systems and GIS raises the question which applications could benefit from such a support and thus document the need for 3D data modeling and data management. For example, for pollution control, in order to keep the maximum load of residents as low as possible, the emission of pollution, exhaust fumes, and noise are simulated during the planning of construction projects. Such simulations are, however, only realistic if they operate on the basis of 3D models. Another similar application area is the calculation of noxious emission after environmental accidents, considered here in the context with geological and meteorological models. For water supply and distribution, the computation of watershed runoff and drainage basins is of interest. For fishery monitoring or simulating, the fish population and its movements are relevant. In geology, soil engineering, and mining, 3D representations give the chance to identify locations where it is worthwhile to exploit ore deposits or oil resources or places that impede test drills due to specific

geological formations or soil strata. A further application is the modeling of geological properties of earthquake areas. Simulations and comparisons can here result in improved prediction methods. Another application is animal observation. Interesting questions relate, for example, to the migration behavior of animals. Frequently, animals like birds or whales are equipped with sensors to pursue their migration routes. If we produce a 2D map from these data, much information gets lost. For example, queries whether whales prefer particular diving depths with specific values for temperature or salt content cannot be answered. In urban planning and urban development, many construction plans are eased by 3D models. A usual 2D map can describe the course of subways and supply mains below the surface only in a limited manner. Possible queries are whether at intersection points a water pipe passes above or below a gas pipe and whether there are soil strata that should not be used for geological reasons. In archeology, many excavation sites are characterized by the fact that these locations have been inhabited for several centuries or even millenniums. Hence, troves of different epochs are located in different soil strata. Since excavations proceed over long periods, it is often difficult to correlate the troves correctly. If we record the troves and the site of the discovery, also later analysis are feasible. The connection of the excavation strata with the use of special materials, or the distribution of similar troves are of particular interest. Models of the archaeological sites in a specific century either with respect to landscape or destroyed buildings can also be designed. In aviation, planes move in the 3D space. Due to improved satellite navigation, pilots are nowadays more flexible in determining their flight routes than some time ago. Example queries are here if there are other planes nearby whose routes have to be taken into account, if there are bad weather fronts that have to be bypassed, what their current shape is, how they will probably develop, and if it is possible to avoid flying over inhabited areas during take-off and landing. If the visibility is low or the terrain is difficult, a “synthetic view” can be computed to ease the orientation and increase

the safety. For telecommunication, in order to be able to position the aerials of mobile phone networks, the wave propagation can be simulated with the aid of 3D models. In cadaster management, efforts are increasingly made to capture cadastral data also in three dimensions. 2D representations are often unsuitable for reproducing the legal situation correctly. This relates both to construction projects above or below the surface (apartment houses, subways) and to issues of pollution control.

2.3 3D Spatial Data Modeling in Databases

2.3.1 Abstract Specifications for 3D Spatial Data

While there is a large variety and divergence of concepts and structures at the discrete level and the implementation level, at the abstract level, we share with these disciplines the identification of 3D points, lines, surfaces, and volumes as the basic 3D spatial objects as well as the use of point set theory and point set topology for formally specifying these objects. 3D spatial objects are modeled as point sets of the Euclidean space $\mathbb{R}^3$ with particular properties. However, we are not aware of any effort to formally specify a general set of 3D spatial data types together with a rather comprehensive collection of 3D operations and predicates in an abstract model. This research with its formal and abstract specification of an algebra for 3D data is meant to be a first step towards this goal.

A widely applied standard specification for geographic features, which most commercial databases like Oracle, mysql and PostgreSQL use to develop their spatial package extensions, is developed by the Open Geospatial Consortium (OGC). OGC is a non-profit organization that deals with conceptual schemes for describing and manipulating the spatial characteristics of geographic features. OGC defines 3D primitives like *GM_Surface* and *GM_Solid* (ISO 19107: Spatial Schema) as the basis for three-dimensional spatial objects. In general, a solid described by OGC is a bounded three-dimensional manifold that has its extent limited by a sequence set of surfaces. Other primitives exist for describing other spatial types like *GM_Curve* and *GM_Point*.

However, the specification does not have rigorous formal definitions for each object it describes, and properties like generality and closure are not considered. Further, since the primitives describe simple geometric features, it is not clear how the complex spatial objects like non-manifold surfaces and volumes with multiple components can be modeled based on the primitives.

2.3.2 Representations of 3D Spatial Data in Databases

As an intermediate step on the way to a treatment of 3D data in GIS, 2.5D representations have been designed like Digital Terrain/Elevation Models[104] (*DTM/DEM*) and Triangulated Irregular Networks[102] (*TIN*). They describe the relief or surface of a terrain by storing for a given point (x, y) of the plane its elevation z. The difference is that DTM uses a regular grid to represent a surface while TIN uses a triangle mesh to represent a surface. An overview of these two approaches can be found in [61]. However, this augmentation is not sufficient to fully represent a 3D object. As an improvement, a unified multidimensional data model is proposed in [26], which allows to store and to describe different dimensional data as well as the relationships between these data.

So far, only a limited number of 3D data models has been proposed for spatial databases and GIS; In general, there are two modeling strategies for representing 3D data in a database context, the topological model and the geometrical model. The topological models assume the full partition of the 3D space (similar to the planar partition in 2D space), and utilize several primitives like nodes, edges and faces to represent such a partition. By explicitly storing the primitives and their corresponding topological relationships in separate predefined relational tables, the elements belonging to a same 3D object can be tracked and reconstructed if needed. A review of the current available topological models can be found in [108]. We confine ourselves here to the 3D Formal Data Structure (3D FDS) [71] and the Tetrahedral Network (TEN) [65]. The Formal Data Structure (3D FDS) [71] provides four primitives node, arc, face,

and edge to support the representation of a full 3D space partition. Four elementary objects, point, line, surface and body can be distinguished. In addition, relationships like node-on-face, arc-on-face, node-in-body, and arc-in-body are also explicitly stored in relational tables to allow singularities. The implementation of this model is proposed in [72]. The Tetrahedral Network(TEN) [2, 65] employs a simplex-oriented approach, in which four primitives, tetrahedron, triangle, arc, and node are defined to represent a full partition of the 3D space. Each primitive is the simplest form in its dimension, and do not overlap with other primitives. The tetrahedron-triangle-edge link is explicitly stored. 3D objects are assembled by these primitives, where a body is composed of tetrahedron, a surface of triangle, a line of arcs and a point of nodes. The topological models rely on relational tables to manage the geometry of a 3D object, thus benefit from the build-in features of a DBMS like indexes and constraints. However, it loses the geometry and expensive table joins and data structure conversions are needed to reconstruct such a geometry.

The geometrical models take an object-oriented approach, thus are more intuitive. The availability of user defined data types (UDT) in most of the commercial databases systems makes it possible to integrate geometrical models into database systems. Different from the topological models that uses simplest geometries as constructive primitives, the geometrical models use more sophisticated primitives to describe 3D objects. A good review of the geometrical models can be found in [58]. One approach is to define a primitive called MultiPolygon to represent a volume object [58]. A MultiPolygon is a collection of 3D polygons that may or may not share boundary. Thus, it can be utilized to define the bounding surface of a volume object. However, it is based on the existing 2D spatial package of mainstream DBMS, e.g. Oracle Spatial, thus is not a real 3D solution. A better approach is to implement polyhedron as the primitive for volume data type in databases. In [3], the author shows how 3D spatial objects can be modeled, i.e. stored, validated and queried, in a Geo-DBMS by using

polyhedron as a primitive. A polyhedron primitive consists of outer shells and inner shells, where a shell is a set of connected polygonal faces that enclose a subset of the 3D space. The polygonal faces consists of outer rings and inner rings, where rings are realized by storing the vertices explicitly (x,y,z) and by describing the arrangement of these vertices in the faces of the polyhedron. A polyhedron represents a simple volume object that is possibly with cavities. However, it has difficulty to represent irregular complex volume objects like non-manifolds and multi-component volumes. Further, the data structure is not suitable for spatial operations like intersection and volume computation. Another attempt is to use tetrahedron as the primitive for volumes. Unlike its correspondents in the topological model that represents tetrahedra with references to triangles, this approach physically stores tetrahedra, and derives triangles and edges whenever needed. Tetrahedron based models are presented in [80, 81]. However, in this approach, the connectivity information among tetraheda needs to be derived, thus to locate neighbors of a tetrahedron becomes very expensive.

2.3.3 Operations on 3D Spatial Data

A few approaches propose operations on basic 3D spatial objects. In [26, 105] besides the geometric set operations intersection, union, and difference, operations are suggested that compute the part of a 3D geometry that is located above another 3D geometry, project a 3D geometry onto a plane, and calculate metric values like the distance, area, and volume of 3D geometries. In [103] the collection of operations proposed contains (i) projection functions from 2D objects to their 3D counterparts, and vice versa, (ii) metric functions like the length, the minimum, maximum, and average gradients and azimuths of 3D polylines, the gradient, azimuth, center of gravity, centroid, surface area, and perimeter of 3D polygons, and the bounding cube, center of gravity, centroid, surface area, and content of polyhedra, and (iii) minimal and maximal distance computations between all types of 3D objects. The work in [3] proposes a polyhedron primitive with operations like point-in-polyhedron test, intersection test,

computation of area, perimeter, volume, bounding box, and centroid, and distance between centroids. Operations of the GeoToolKit [6] include distance, surface area, and volume functions, topological predicates testing for intersection and containment, the intersection operation, and the projection onto a given plane.

In summary, despite little nuances, there seems to be a consensus about the kind of 3D objects that should be supported in a model for 3D spatial data. However, it is unclear how these object types should be defined, what properties they should have, which operations and predicates should be defined on them, and how to represent these data types. It is the goal of this research work to provide the solutions.

2.4 Qualitative Spatial Relationships between 3D Spatial Data

Research on qualitative spatial relationships has had a long tradition in spatial databases, Geographic Information Systems (GIS), and other disciplines like cognitive science, robotics, and artificial intelligence. Topological relationships and cardinal directions relationships are the two most important qualitative spatial relationships that characterize the relative position of spatial objects like inside, meet, north, and west. From a database and GIS perspective, the exploration of these relationships between spatial objects has been motivated by the need of formally defined topological predicates and directional predicates as filter conditions for spatial selections and spatial joins in spatial query languages and as a support for spatial data retrieval and analysis tasks. So far, most efforts have focused on modeling spatial relationships between objects in 2D space and the models in the 3D space have been rare.

2.4.1 Topological Relationships

A special emphasis of spatial research has been put on the exploration of topological relationships (for example, overlap, inside, disjoint, meet) between spatial objects. They describe purely qualitative properties that characterize the relative positions of spatial objects towards each other and that are preserved under certain continuous transformations including all affine transformations. From a database and

GIS perspective, their investigation has been motivated by the necessity of formally defined topological predicates as filter conditions for spatial selections and spatial joins. Extensive study has been performed on the topological relationships between two spatial objects in two-dimensional space. An important approach for characterizing them rests on the so-called 9-intersection model, which employs point set theory and point set topology [32]. The model is based on the nine possible intersections of the boundary (∂A), interior (A°), and exterior (A^-) of a spatial object A with the corresponding components of another object B. Each intersection is tested with regard to the topologically invariant criteria of emptiness and non-emptiness. The topological relationship between two spatial objects A and B can be expressed by evaluating the matrix in Figure 2-1. A total of $2^9 = 512$ different configurations are possible from which only a certain subset makes sense depending on the combination of spatial data types just considered. A complete collection of mutually exclusive topological relationships can be determined for each combination of simple spatial data types [33] and, as a generalization, for each combination of complex spatial data types [88].

A topic that has been partially formally explored at the abstract level deals with topological relationships between simple 3D spatial objects. In [31], the author applies the 9-intersection model to simply-connected 3D spatial objects, that is, simple 3D lines, simple surfaces (no holes), and simple volumes (no cavities), in order to determine their topological relationships. Table 2-1 shows the numbers of topological relationships identified. Zlatanova also investigated the possible 3D topological relationships in [106] by developing a complete set of negative conditions. The 9-intersection model for 3D

$$\begin{pmatrix} A^\circ \cap B^\circ \neq \varnothing & A^\circ \cap \partial B \neq \varnothing & A^\circ \cap B^- \neq \varnothing \\ \partial A \cap B^\circ \neq \varnothing & \partial A \cap \partial B \neq \varnothing & \partial A \cap B^- \neq \varnothing \\ A^- \cap B^\circ \neq \varnothing & A^- \cap \partial B \neq \varnothing & A^- \cap B^- \neq \varnothing \end{pmatrix}$$

Figure 2-1. The 9-intersection matrix for topological relationships.

Table 2-1. Number of topological predicates between two simple 3D spatial objects in the Euclidean plane $\mathbb{R}^3$

	simple 3D point	simple 3D line	simple surface	simple volume
simple 3D point	2	3	3	3
simple 3D line	3	33	29	19
simple surface	3	29	41	19
simple volume	3	19	19	8

can be extended with the dimensionality being considered. In [14], the values of the matrix elements are extended. Besides the $\varnothing$ and $\neg\varnothing$ symbols, the $*$ is used to indicate the no specification for the resulting set at this position, and the numbers (0,1,2,3) refer to the dimensionality of the resulting set. Therefore, unlike the 1:1 mapping between the matrix with topological relationships in 9-intersection model, a matrix that contains a $*$ value represents a class of topological relationships. As a result, the topological relationships are clustered and manageable to the user. However, the focus has been only on the simple 3D spatial objects, the exploration of valid 9IMs for complex spatial objects is still left open, and this is our goal to bridge this gap.

Above two models in common apply the set theory to identify topological relationships. Thus, they can be categorized to point-set based topological relationships models. There are also other approaches that do not employ the point-set theory. The approach in [82] investigates topological predicates between cell complexes, which are structures from algebraic topology. It turns out that, due to limitations of cell complexes, the topological relationships between them are only a subset of the derived point-set based topological relationships. The topological relationships between 3D spatial objects that consist of a serial of cell complexes can be described by the combination of relationships between those cell [48]. Dimension Model is a model that is independent of the 9-intersection model. It defines dimensional elements on a spatial object, and all the dimensional elements contribute to final result. Three levels of details for the topological relationships are developed for different application purposes. The dimension model can

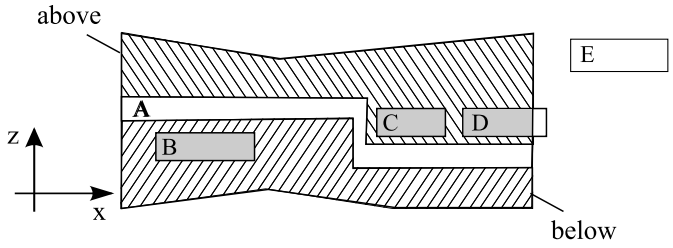


Figure 2-2. An example of the projection-based directional model.

distinguish some cases, especially meet cases, that the 9-intersection model can not identify. However, since it leaves the abstract topological space where only point sets are used, it is not clear how the dimensional elements can be constructed.

2.4.2 Cardinal Direction Relationships

Besides topological relations, cardinal directions have turned out to be an important class of qualitative spatial relations. It describes absolute directional relationships like north and southwest with respect to a given reference system in contrast to relative directional relationships like front and left; thus, cardinal directions describe an order in space. Similar to topological relations, in spatial databases they are also relevant as selection and join conditions in spatial queries. Hence, the determination of and reasoning with cardinal directions between spatial objects is an important research issue. For a long time the research focus has been on the cardinal direction relations between 2D spatial objects. Cardinal direction models for 3D spatial objects are rare. Most models developed are points-based models so that the extent of objects is ignored. An absolute direction model that considers cardinal directions like north and east can be found in SLC99GEOINF. It ignores the elevation and projects the space to a 2D coordinate system. In LLL08CSSE, a conceptual framework that depicts positional relationships including both distance and direction relationships is developed. For all the mentioned models, spatial objects are generalized as points, and directions are based on point locations so that the models become rather imprecise.

The projection-based directional model [14] extrudes a reference object along the coordinate axes in the direction requested. The extrusion body is tested for the intersection with a target object. The model enables the definition of the semantics of a set of cardinal directions such as *eastOf*, *westOf*, *northOf*, *southOf*, *above* and *below*. Figure 2-2 shows an example. The two extrusion bodies of the reference object A are identified as *above* and *below*. The target object B intersects the extrusion body *below* so that B is below A. The target objects C and D intersect the extrusion body *above* so that they are above A. The target object E does not intersect any extrusion body so that no statement can be made. Obviously, the expressiveness is a major weakness of this model since the model cannot determine the cardinal direction for all spatial configurations.

The three-dimensional cardinal direction (*TCD*) model [20] partially considers the shape of objects. Based on the direction relation matrix (*DRM*) model [44] for cardinal relations in the two-dimensional space, the TCD model takes the minimum bounding box (cube) of a reference object, extends its boundary faces to planes, and divides the space into 27 direction tiles, as shown in Figure 2-3A. The superscripts U, B and M denote the top, middle, and bottom of the reference object, respectively. This model leads to the 27 atomic TCD relations, $NW^U, \ldots, SE^U, NW^M, \ldots, SE^M, NW^B, \ldots, SE^B$ that symbolize the basic cardinal direction relations between the components of the target object and the reference object. The TCD relations form a set of exhaustive

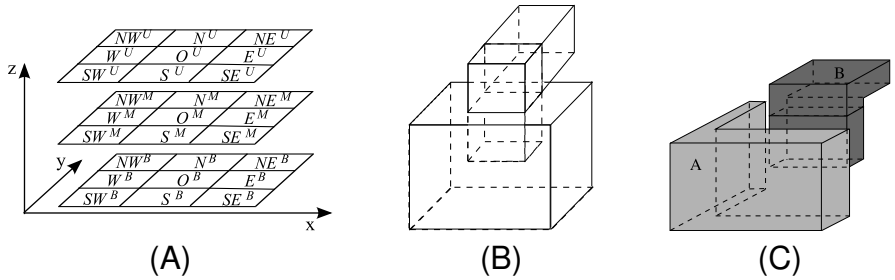


Figure 2-3. The TCD relation model. The 27 atomic TCD relations (A), and two scenarios where the cardinal direction between two objects A and B are both $dir(A, B) = N^U : O^U : O^M$ (B,C).

and pairwise disjoint relations. Figure 2-3B shows an example of two volumes A and B. If A is taken as the reference object, then the direction relationship between A and B is $dir(A, B) = N^U : O^U : O^M$. The TCD model presents a major improvement on expressiveness. However, the shape of the reference object does not contribute to the determination of final result, thus may lead to the loss of precision and sometimes incorrect results. For instance, in Figure 2-3C, the model yields $dir(A, B) = N^U : O^U : O^M$ as the direction relationships between objects A and B. But the direction O^U should not be part of direction relationships since no part of B lies directly above A. Another problem with this model is the converseness problem, which leads to unintuitive cardinal relations. For example, the model yields $dir(B, A) = W^M : O^M : E^M : SW^M : S^M : SE^M : W^B : O^B : E^B : SW^B : S^B : SE^B$ in Figure 2-3B. But this is unequal to the expected inverse $S^B : O^B : O^M$ of $dir(A, B)$.

CHAPTER 3
ABSTRACT THREE-DIMENSIONAL DATA MODELING

An abstract model of the three-dimensional (3D) data in spatial databases provides rigorous and comprehensive specification of 3D spatial data and 3D operations in terms of what the nature of the 3D data is, how to construct such spatial objects, and what the semantics of the operations are. One must understand that, when developing a data model, it is important to start with an abstract model whose focus is on completeness, closure, consistency and genericity of the type definitions and the semantics of operations, rather than the performance issues like efficiency and scalability of the data structures and algorithms.

In this chapter, we introduce in details an abstract data model for 3D data, called Spatial Algebra 3D at the Abstract Level (*SPAL3D-A*). We first present the definitions of 3D spatial data types and the semantics of corresponding operations in Section 3.1. Then in Section 3.2, we discuss two major qualitative spatial relationships between 3D spatial objects, which is motivated by the need of spatial predicates in the database query language.

3.1 An Abstract Model of Three-Dimensional Spatial Data Types

We now come to the core of the Spatial Algebra 3D at the Abstract Level (*SPAL3D-A*), the definitions of the three-dimensional (3D) spatial data types and operations. We first describe some design considerations for 3D spatial data types in Section 3.1.1. The remaining sections focus on the rigorous definition of each single data type (Section 3.1.2) and the operations on them (Section 3.1.3).

3.1.1 Design Considerations

The development of our 3D spatial data types and SPAL3D as a whole is influenced by several design criteria which we summarize in this subsection.

The first design criterion refers to the generality and versatility of the specified 3D spatial data types. The types should be defined as general as possible but not

go beyond spatial reality. They should be application-neutral, i.e., they should not be restricted to a particular application but applicable to a broad and diverse spectrum of application domains.

A second design criterion is the fulfilment of closure properties. This means that the types should be closed under the operations of geometric union, intersection, and difference. The result of each such operation should yield a well-defined object that corresponds to the definition of the spatial data types. The benefit is that possible geometric anomalies like dangling or missing pieces are avoided. For example, the intersection of two volumes should be a volume again; dangling and missing points, lines, and surfaces should not exist.

A third design criterion relates to a rigorous definition of the data types. Their semantics, which determines the admissible point sets for their objects, requires a formal, precise, and unique definition in order to avoid ambiguities both for the user and the implementor. Hence, we design an abstract model first. Uniqueness especially implies that each object has a unique representation, even if several different representations would yield the same point set for an object.

A fourth design criterion is the extensibility of the type system of SPAL3D. It is common consensus in the spatial database and GIS community that a complete and closed set of spatial data types and operations cannot be determined. Even though a designer may provide a good collection of types and operations, there will always be applications requiring additional operations on available types or new types with new operations. A type system for SPAL3D must therefore be extensible for new types, operations, and predicates.

For each data type we give both an unstructured and a structured definition. The unstructured definition determines the point set view of a spatial object, i.e., such an object is specified as a pure point set. The structured definition gives an equivalent, unique representation and emphasizes the component view of a 3D spatial object. In

general, objects of all 3D spatial data types may only include a finite number of disjoint components. In addition, surfaces may have holes, and volumes may incorporate cavities. The formal framework for both views leverages point set theory and point set topology [41]. We assume that the reader is familiar with their basic concepts like topological and metric spaces. For an object A of each 3D spatial data type, we will distinguish different object components and specify the topological notions of boundary (∂A), interior (A°), exterior (A^-), and closure $(\overline{A})$. These notions are later needed, e.g., for the computation of topological relationships between 3D spatial objects.

3.1.2 Three-dimensional Spatial Data Types

3.1.2.1 Mathematical Background: Point Set Topology

Our definitions of the spatial data types in three-dimensional space are based on point set theory and point set topology [42]. Spatial data are embedded into the three-dimensional Euclidean space $\mathbb{R}^3$ and modeled as infinite point sets. Before we present the unstructured and structured definitions of the data types, we introduce a few needed mathematical notations as well as some topological and metric concepts.

Let $\mathbb{N}$ be the set of natural numbers, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, $\mathbb{R}$ be the set of real numbers, $\mathbb{R}^+ = \{x \in \mathbb{R} \,|\, x > 0\}$, and $\mathbb{R}^n$ with $n \in \mathbb{N}$ be the n-dimensional Euclidean space. We assume the existence of a Euclidean distance function $d : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ with $d(p, q) = d((x_1, y_1, z_1), (x_2, y_2, z_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$. Then for any point $p \in \mathbb{R}^3$ and $r \in \mathbb{R}^+$, the set $N_r(p) = \{q \in \mathbb{R}^3 \,|\, d(p, q) \leq r\}$ is called the (closed) neighborhood with radius r and center p. We can think of the neighborhood around p as a closed ball containing all of the points in $\mathbb{R}^3$ that are "near" p.

Let $X \subset \mathbb{R}^3$ and $p \in \mathbb{R}^3$. p is an interior point of X if there exists a neighborhood $N_r(p)$ such that $N_r(p) \subseteq X$. p is an exterior point of X if there exists a neighborhood $N_r(p)$ such that $N_r(p) \cap X = \emptyset$. p is a boundary point of X if p is neither an interior nor exterior point of X. p is a closure point of X if p is either an interior or boundary point of X. The set of all interior / exterior / boundary / closure points of X is called the *interior* /

exterior / *boundary* / *closure* of X and is denoted by X° / X^- / ∂X / $\overline{X}$. A point p is a limit point of X if for every neighborhood $N_r(p)$ holds that $(N_r(p) - p) \cap X \neq \emptyset$. X is called an open set in $\mathbb{R}^3$ if $X = X^\circ$. X is called a closed set in $\mathbb{R}^3$ if every limit point of X is a point of X. It follows from the definition that every interior point of X is a limit point of X. Thus, limit points need not be boundary points. The converse is also true. A boundary point of X need not be a limit point; it is then called an isolated point of X. For the closure of X we obtain that $\overline{X} = \partial X \cup X^\circ$.

For the specification of the spatial data types, definitions are needed for bounded and connected sets. Two subsets $X, Y \subseteq \mathbb{R}^3$ are said to be separated if, and only if, $X \cap \overline{Y} = \varnothing = \overline{X} \cap Y$. A set $X \subseteq \mathbb{R}^3$ is connected if, and only if, it is not the union of two nonempty separated sets. A set $X \subset \mathbb{R}^3$ is said to be bounded if there is an $r \in \mathbb{R}^+$ such that $||p|| < r$ for all $p = (x, y, z) \in X$ where $||p|| = \sqrt{x^2 + y^2 + z^2}$ is the norm or length of p.

We now introduce another important relevant concept homeomorphism or topological equivalent which will be used extensively in our definitions for spatial data types. Homeomorphism is an important concept that is widely used in both mathematics and topology to classify geometries from a topological viewpoint. We present the definition of homeomorphism in Definition 1.

Definition 1. *A function between topological spaces $f : X \to Y$ is called a homeomorphism if f has following properties:*

(i) f is a bijection (one to one and onto);

(ii) f is continuous;

(iii) the inverse function f^{-1} is continuous.

If a homeomorphism $f : X \to Y$ exists, we say that X and Y are homeomorphic.

A homeomorphism gives a one-to-one correspondence between points in topological spaces X and Y. It is the mapping which preserves all the topological properties of a given topological space. To describe this, we often say that the

homeomorphic spaces have the same topology, or are topologically equivalent. If two topological spaces are topologically equivalent, they are indistinguishable by topological methods, so from a topological viewpoint they are the same. Roughly speaking, a topological set is a geometric object, and the homeomorphism is a continuous tilting, translating, bending and twisting of the object into a new geometric shape. Thus, a square and a circle are homeomorphic to each other, but a sphere and a doughnut are not.

Finally, we introduce the definitions for the simplest geometric shapes of every dimension in Definition 2. We call these shapes the n-dimensional disk denoted as D^n.

Definition 2. *Let $r \in \mathbb{R}^+$, then we refer to the set $B_r(O) = \{p \in \mathbb{R}^n \mid \|p\| \leq r\}$ as the n-dimensional disk with radius r or sometimes the n-dimensional ball, denoted D^n. The boundary of the n-dimensional disk D^n, denoted as ∂D^n, is defined to be the set $\partial B_r(O) = \{p \in \mathbb{R}^n \mid \|p\| = r\}$. The* interior *of the n-dimensional disk D^n, denoted as $(D^n)^\circ$, is defined to be the set $B_r(O)^\circ = \{p \in \mathbb{R}^n \mid \|p\| < r\}$.*

As a result, a circle is a 2-dimensional disk D^2 and a sphere is a 3-dimensional disk (ball) D^3. These simplest geometric shapes in n dimension are useful in defining more complex spatial objects.

3.1.2.2 The Data Type *point3D*

Three-dimensional points are the origin of the reduction and generalization process to 2D points and the zero-dimensional objects in the 3D space. Those applications benefit from 3D points which besides their positions are especially interested in their heights. For example, if we consider all lighthouses at the coasts of the U.S., the heights of their beacons over sea level (together with other factors like the luminosity of the beacons and the curvature of the Earth's surface) allow us to geometrically compute the spatial range of the beacons, the sea area covered by all beacons, and the information about whether this sea area is without gaps. This example also illustrates that from an

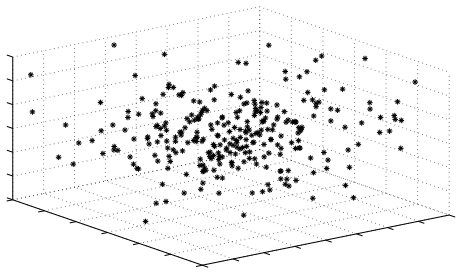


Figure 3-1. A *point3D* object.

application perspective a 3D point object should be able to incorporate a finite set of single 3D points.

Definition 3 specifies a value of type *point3D* as a finite set of single or isolated points in the 3D space. Unstructured and structured definitions coincide for this type.

Definition 3. *The spatial data type point3D is defined as*

$$\textit{point3D} = \{P \subset \mathbb{R}^3 \mid P \text{ is finite}\}$$

We call a value of this type complex point. If $P \in$ point is a singleton set, i.e., $|P| = 1$, P is denoted as a simple point. If $|P| = 0$, i.e., $P = \varnothing$, P is called empty point. The topological notions of boundary, interior, exterior, and closure of a point3D object are defined as follows. For a simple point p, we specify $\partial p = \varnothing$ and $p^\circ = p$. For a complex point $P = \{p_1, \ldots, p_n\}$, we obtain $\partial P = \varnothing$, $P^\circ = \bigcup_{i=1}^{n} p_i^\circ$, $P^- = \mathbb{R}^3 - (\partial P \cup P^\circ) = \mathbb{R}^3 - P^\circ$, and $\overline{P} = \partial P \cup P^\circ = P^\circ$.

Definition 3 states that type *point3D* (see an example in Figure 3-1) contains all finite sets of the power set of $\mathbb{R}^3$. The empty point $\varnothing$ is the identity of geometric union and is admitted since it can be the result of a geometric operation, e.g., if a point object has nothing in common with another point object in a geometric intersection operation.

3.1.2.3 The Data Type *line3D*

Three-dimensional lines are the origin of the reduction and generalization process to 2D lines and the one-dimensional objects in the 3D space. Again, the involvement of the third dimension, i.e., the height, makes the fundamental difference. For example,

if we consider a road in the mountains as a 3D curve, this curve has rather different properties than its 2D counterpart. The length computation will lead to different values. The 3D line geometry of a road enables calculations like the maximum, minimum, or average height of the road as well as the maximum or minimum slope. This makes it possible, e.g., to capture the 3D geometry of the U.S. bus network in a single 3D line object and to intersect it with another single 3D line object modeling the U.S. taxi network. For these 3D objects, this section contains the definition of the spatial data type *line3D*.

Before we present the unstructured and structured definitions of this data type, we introduce a few needed mathematical notations as well as some topological and metric concepts. Let $\mathbb{N}$ be the set of natural numbers, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, $\mathbb{R}$ be the set of real numbers, $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$, and $\mathbb{R}^n$ with $n \in \mathbb{N}$ be the n-dimensional Euclidean space. For two points $p = (x_1, y_1, z_1), q = (x_2, y_2, z_2) \in \mathbb{R}^3$, we assume the existence of the Euclidean distance function $dist : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ with $dist(p, q) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$. With this notion of distance, we can proceed to define what is meant by a neighborhood of a point in $\mathbb{R}^3$. For $p \in \mathbb{R}^3$ and $\epsilon \in \mathbb{R}^+$, the set $N_\epsilon(p) = \{q \in \mathbb{R}^3 \mid dist(p, q) < \epsilon\}$ is called the open neighborhood(open disk, open ball) with radius ϵ and center p. The set $\overline{N}_\epsilon(p) = \{q \in \mathbb{R}^3 \mid dist(p, q) \leq \epsilon\}$ is called the closed neighborhood with radius ϵ and center p. Any open (closed) neighborhood with center p is denoted by $N(p)$ ($\overline{N}(p)$).

We are now able to define the notion of a continuous mapping which preserves neighborhood relations between mapped points in two metric spaces. In Definition 4 we are especially interested in the continuity of mappings f with the signature $f : \mathbb{R} \to \mathbb{R}^3$.

Definition 4. *Let $X \subset \mathbb{R}$ and $f : X \to \mathbb{R}^3$ be a mapping. Then f is said to be continuous at a point $x_0 \in X$ if for all $\epsilon \in \mathbb{R}^+$ there exists a $\delta \in \mathbb{R}^+$ such that for all $x \in N_\delta(x_0) \cap X$ we obtain that $f(x) \in N_\epsilon(f(x_0))$. The mapping f is said to be continuous on X if it is continuous at every point of X.*

Definition 5 introduces the notions of a connected set, a path-connected set, and a bounded set in $\mathbb{R}^3$.

Definition 5. *Two subsets $X, Y \subseteq \mathbb{R}^3$ are said to be separated if $X \cap \overline{Y} = \varnothing = \overline{X} \cap Y$. A set $X \subseteq \mathbb{R}^3$ is connected if it is not the union of two nonempty separated sets. A set $X \in \mathbb{R}^3$ is path-connected if for all points $x, y \in X$ there is a continuous mapping $f : [0,1] \to X$ such that $f(0) = x$ and $f(1) = y$ holds. A set $X \subset \mathbb{R}^3$ is said to be bounded if there is an $r \in \mathbb{R}^+$ such that $||p|| < r$ for all $p = (x, y, z) \in X$ where $||p|| = \sqrt{x^2 + y^2 + z^2}$ is the norm or length of p.*

Example 3.1.0 addresses the important difference between path-connectedness and connectedness.

Example 3.1.0. *Path-connectedness implies connectedness but not vice versa. For example, let us consider the topologist's curve (see Figure 3-2) defined as the set $A = \{(x, \sin\frac{\pi}{x}) \mid x > 0\}$. The closure of this set is connected but not path-connected. Its boundary is the set $\{(0, z) \mid -1 \leq z \leq 1\}$ but there is no continuous path within the closure that joins points of the interior (i.e., the curve itself) with points of the boundary. Although A does not reach the y-axis, it shows a "bad behavior" near that axis because it oscillates an unbounded number of times as it approaches. Such an unbounded number of oscillations cannot be later mapped to a finite representation in an implementation; hence, we are interested in excluding this case from our object representations. The requirement of path-connectedness of a point set excludes this case. The deeper reason is that each path-connected set is semi-analytic [70] and that*

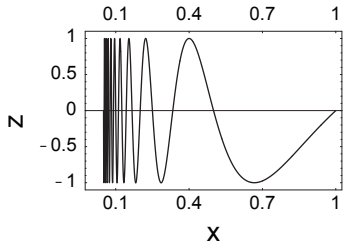


Figure 3-2. The topologist's curve $(x, \sin\frac{\pi}{x})$ for $x > 0$.

this is not the case for connected sets. Note that the set $A_r = \{(x, \sin\frac{\pi}{x}) \,|\, x \geq r > 0\}$ *is path-connected; it oscillates a finite number of times as it approaches* $x = r$*.* □

Finally, for a function $f : Xs \to Y$ and a set $A \subseteq X$ we introduce the notation $f(A) = \{f(x) \,|\, x \in A\}$. Definition 6 gives us an unstructured definition for 3D lines as the union of the images of a finite number of continuous mappings.

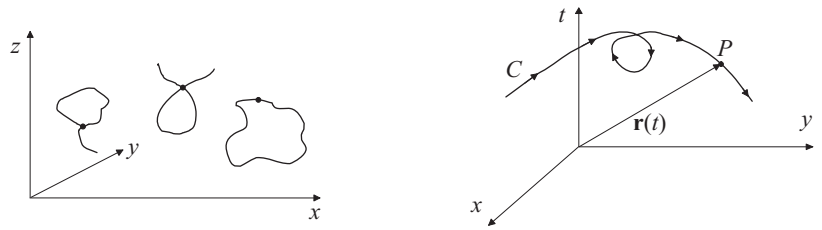
Definition 6. *The spatial data type line3D is defined as*

$$\begin{aligned} \textit{line3D} = \{L \subset \mathbb{R}^3 \,|\, &\text{(i)} \quad L = \textstyle\bigcup_{i=1}^{n} f_i([0,1]) \text{ with } n \in \mathbb{N}_0 \text{ and } f_i : [0,1] \to \mathbb{R}^3 \\ &\text{(ii)} \ \ \forall\, 1 \leq i \leq n : f_i : [0,1] \to \mathbb{R}^3 \text{ is a continuous mapping} \\ &\text{(iii)} \ \forall\, 1 \leq i \leq n : f_i([0,1]) \text{ is a bounded and path-connected set} \\ &\text{(iv)} \ \forall\, 1 \leq i \leq n : |f_i([0,1])| > 1\} \end{aligned}$$

We call a value of this type complex line. For $n = 1$*, we obtain a simple line; for* $n = 0$*, we get the empty line* $\varnothing$ *indicating the empty point set.*

Condition (i) defines a complex 3D line object as the union of the images of a finite number of mappings from the one-dimensional interval $[0, 1]$ to the 3D space. Condition (ii) requires that each of these functions is continuous according to Definition 4. Condition (iii) only admits "well behaving" point sets as images of these functions. Condition (iv) avoids degenerate line objects consisting of a single point. Images of different functions may intersect each other. The same line object can be represented by different collections of functions. Figure 3-3A shows a complex 3D line that can be described as the union of the images of six continuous mappings according to the unstructured view.

Example 3.1.0. *To understand the definition better, it is worthwhile to have a closer look at a single image* $l([0, 1])$*. A vector-valued function or vector function* ***r*** *can represent such an image. It is a function whose domain is the set of real numbers and whose range is a set of 3D vectors. This means that for every number* t *in the domain of* ***r*** *there is a unique vector in* V_3 *(the set of all 3D vectors) denoted by* $\mathbf{r}(t)$*. If* $f(t)$*,* $g(t)$*, and* $h(t)$



(A) (B)

Figure 3-3. A complex 3D line object example. A complex 3D line that can be described as the union of the images of six continuous mappings (thick points indicate the start and end points of the f_i) in the unstructured view and as three blocks in the structured view (A), and a space curve as a component of a *line3D* object (B).

are the components of the vector $\mathbf{r}(t)$*, then* f*,* g*, and* h *are real-valued functions called the component functions of* $\mathbf{r}$*, and we can write*

$$\mathbf{r}(t) = \langle f(t), g(t), h(t) \rangle = f(t)\,\mathbf{i} + g(t)\,\mathbf{j} + h(t)\,\mathbf{k}$$

where $\mathbf{i} = \langle 1, 0, 0 \rangle$*,* $\mathbf{j} = \langle 0, 1, 0 \rangle$*, and* $\mathbf{k} = \langle 0, 0, 1 \rangle$ *are the standard basis vectors in* V_3*. If* f*,* g*, and* h *are continuous, then* $\mathbf{r}$ *is also continuous. We can observe a close connection between continuous vector functions and line3D objects. Suppose* f*,* g*, and* h *are continuous real valued functions on* $[0, 1]$*. Then the set* $C = \{(x, y, z) \in \mathbb{R}^3 \mid t \in [0, 1], x = f(t), y = g(t), z = h(t)\}$ *describes a space curve, i.e., a component of a line3D object. If we consider the vector function* $\mathbf{r}(t) = \langle f(t), g(t), h(t) \rangle$*, then* $\mathbf{r}(t)$ *is the position vector of the point* $P = (f(t), g(t), h(t))$ *on* C*. Thus, any continuous vector function* $\mathbf{r}$ *defines a space curve* C *that is traced out by the tip of the moving vector* $\mathbf{r}(t)$*, as shown in Figure 3-3B. Obviously,* $\mathbf{r}$ *is bounded and path-connected.* □

For a structured definition, we separate the point set of a *line3D* object into appropriate components called curves. A curve, which is formally defined in Definition 7, is considered a non-self-intersecting 3D line and thus a restricted space curve. This restriction could be relaxed but has the advantage that each complex 3D line can be uniquely described as a collection of non-self-intersecting 3D lines.

Definition 7. *The set curves of curves over* $\mathbb{R}^3$ *is defined as*

$$
\begin{aligned}
\mathit{curves} = \{ f([0,1]) \mid\ & \text{(i)}\quad f : [0,1] \to \mathbb{R}^3 \text{ is a continuous mapping} \\
& \text{(ii)}\quad f([0,1]) \text{ is a bounded and path-connected set} \\
& \text{(iii)}\ \forall\, a, b \in]0,1[: a \neq b \Rightarrow f(a) \neq f(b) \\
& \text{(iv)}\ \forall\, a \in \{0,1\}\ \forall\, b \in]0,1[: f(a) \neq f(b) \\
& \text{(v)}\quad |f([0,1])| > 1 \}
\end{aligned}
$$

The values $f(0)$ and $f(1)$ are called the end points of a curve. If $f(0) = f(1)$, we call the curve closed or a loop.

This definition forbids self-intersecting (condition (iii)) and self-touching (condition (iv)) curves, i.e., the equality of different interior points and the equality of an interior point with an end point. As an important special case, we mention curves that describe straight line segments in the 3D space.

For a unique definition of complex lines from a set of curves, we will have to require that these curves do not share any points except common end points. For this purpose, in Definition 8, we specify the predicates quasi-disjoint and meet on curves.

Definition 8. *Let $A \subseteq \mathit{curves}$, and let $l_1, l_2 \in A$ be defined by the functions f_1 and f_2. Then*

$$
\begin{aligned}
l_1 \text{ and } l_2 \text{ are quasi-disjoint} \quad \Leftrightarrow\quad & \forall\, a, b \in]0,1[: f_1(a) \neq f_2(b) \wedge \\
& \forall\, a \in \{0,1\}\ \forall\, b \in]0,1[: \\
& \qquad (f_1(a) \neq f_2(b) \ \wedge\ f_2(a) \neq f_1(b)) \wedge \\
& \qquad \neg((f_1(0) = f_2(0) \ \wedge\ f_1(1) = f_2(1)) \vee \\
& \qquad (f_1(0) = f_2(1) \ \wedge\ f_1(1) = f_2(0))) \\
l_1 \text{ and } l_2 \text{ meet} \quad \Leftrightarrow\quad & l_1 \text{ and } l_2 \text{ are quasi-disjoint} \wedge \\
& \exists\, a, b \in \{0,1\} : f_1(a) = f_2(b) \\
l_1 \text{ and } l_2 \text{ are connected in } A \quad \Leftrightarrow\quad & \exists\, c_1, \ldots, c_n \in A : l_1 = c_1, l_2 = c_n, \text{ and} \\
& \forall\, i \in \{1, \ldots, n-1\} : c_i \text{ and } c_{i+1} \mathit{meet}
\end{aligned}
$$

Due to the uniqueness constraint (see Section 3.1.1), the definition of quasi-disjoint requires that two curves do not share any interior points, an end point of one curve does not coincide with an interior point of the other curve, and both curves do not form a loop

together. The reason for forbidding such composite loops is that a loop can be described by a single function. Note that two curves that are both loops and meet in a common end point are quasi-disjoint. Two curves meet if they share exactly and only one end point. Two curves in a set are connected if they can be linked by a chain of curves of that set.

Next, in Definition 9, we introduce the concept of a block specifying a connected component of a complex line.

Definition 9. *The set blocks of blocks over the set curves is defined as*

$$
\begin{aligned}
\mathit{blocks} = \{\textstyle\bigcup_{i=1}^{m} l_i \mid (\mathrm{i}) \quad & m \in \mathbb{N}, \forall\, 1 \leq i \leq m : l_i \in \mathit{curves} \\
(\mathrm{ii}) \quad & \forall\, 1 \leq i < j \leq m : l_i \text{ and } l_j \text{ are quasi-disjoint} \\
(\mathrm{iii}) \quad & \forall\, 1 \leq i < j \leq m : l_i \text{ and } l_j \text{ are } \textit{connected} \\
(\mathrm{iv}) \quad & \forall\, p \in \textstyle\bigcup_{i=1}^{m} \{f_i(0), f_i(1)\} : \\
& \mathit{card}(\{f_i \mid 1 \leq i \leq m \ \wedge\ f_i(0) = p \ \wedge\ f_i(1) = p\}) \neq 0 \ \vee \\
& \mathit{card}(\{f_i \mid 1 \leq i \leq m \ \wedge\ ((f_i(0) = p \ \wedge\ f_i(1) \neq p) \ \vee \\
& \qquad (f_i(0) \neq p \ \wedge\ f_i(1) = p))\}) \neq 2\}
\end{aligned}
$$

Two blocks $b_1, b_2 \in \mathit{blocks}$ *are disjoint if, and only if,* $b_1 \cap b_2 = \varnothing$.

Condition (i) states that a block consists of a collection of curves. Condition (ii) expresses that these curves may at most share end points. The connectedness of the curves in a block is formulated in condition (iii). Condition (iv) ensures uniqueness of representation. Uniqueness is guaranteed if each end point either has one or more incident closed curves, or, in case there is no incident closed curve, does not only have two incident non-closed curves. That latter case aims to avoid meeting curves that could be described by a single function.

We are now able to give the structured definition for complex 3D lines and also specify the boundary, interior, and exterior of a complex line.

Definition 10. *The spatial data type line3D is defined as*

$$\textit{line3D} = \{\bigcup_{i=1}^{n} b_i \,|\, (\mathrm{i})\ \ n \in \mathbb{N}, \forall\, 1 \leq i \leq n : b_i \in \textit{blocks}$$
$$(\mathrm{ii})\ \forall\, 1 \leq i < j \leq n : b_i \text{ and } b_j \text{ are disjoint}\}$$

We call a value of this type complex line. For $n = 0$*, we obtain the empty line* $\varnothing$*. Let* $L \in$ *line3D and, based on Definition 6, let* $E(L) = \bigcup_{i=1}^{n}\{f_i(0), f_i(1)\}$ *be the set of end points of all* n *curves of* L*. We define the boundary of* L *as*

$$\partial L = E(L) - \{p \in E(L) \,|\, card(\{f_i \,|\, 1 \leq i \leq m \ \wedge\ f_i(0) = p\}) +$$
$$card(\{f_i \,|\, 1 \leq i \leq m \ \wedge\ f_i(1) = p\}) \neq 1\}$$

The closure of L *is defined as* $\overline{L} = L$*. The interior of* L *is given as* $L^{\circ} = \overline{L} - \partial L = L - \partial L$*, and the exterior is defined as* $L^{-} = \mathbb{R}^3 - L$*, since* $\mathbb{R}^3$ *is the embedding space.*

Definition 10 implies that no curve in a block may share a point (i.e., intersect, touch, meet) with a curve in any other block. The boundary of a 3D line L is the set of its end points minus those end points that are shared by several curves. The shared points belong to the interior of a 3D line. The end points of closed curves are counted twice so that the end point of a single closed curve forming a *line3D* object does not belong to the boundary (but to the interior). The closure of L is L itself including its end points. Figure 3-3A shows a complex line consisting of three blocks according to the structured view.

3.1.2.4 The Data Type *Surface*

Our definition of surfaces is based on point set theory and point set topology. Surfaces are embedded into the three-dimensional Euclidean space $\mathbb{R}^3$ and are modeled as special infinite point sets. In order to formally characterize such a point set, we need to introduce some additional concepts.

We have introduced the concept of neighborhood of a point p that describes all points that are “near” p in Euclidean space R^3. When p belongs to a subset $X \subseteq \mathbb{R}^3$, we also want to identify all points in X that are “near” the point p. Therefore, we introduce the concept local neighborhood in Definition 11.

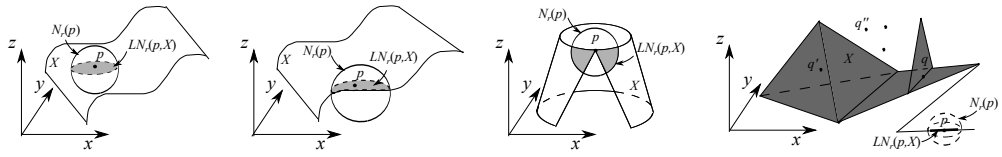


Figure 3-4. The subset $LN_r(p, X)$ is a local neighborhood of the point p in subset X in (A),(B),(C) and (D).

Definition 11. *Let X be a subset in the three-dimensional Euclidean space $\mathbb{R}^3$ ($X \subseteq \mathbb{R}^3$), p be a point in X and $N_r(p)$ be the neighborhood with radius r and center p, then the set $LN_r(p, X) = N_r(p) \cap X$ is called the local neighborhood of p in X.*

With the concept of local neighborhood, we can now define a surface. A simple form of a surface is a set X where for any point $p \in X$, there exist a local neighborhood $LN_r(p, X)$ such that $LN_r(p, X)$ is homeomorphic to a 2D disk (D^2) [42]. In other word, a simple form of a surface is a set where everywhere is locally homeomorphic to a closed 2D disk (Figure 3-4A). We call such surfaces the simple manifold surfaces and we give the detailed definition later. However, in the real world, a surface object is more complex, which may contain points where such a homeomorphism to a 2D disk does not exist. For example, the set X in Figure 3-4C is intuitively a surface object, but it is not homeomorphic to a 2D disk at point p. Thus, we aim at defining the most general form of a surface object that is bounded, does not contain any volume part, tangling lines or tangling points, and possibly contains multiple surface components. For this purpose, we first define a surface point.

Definition 12. *Let $X \subseteq \mathbb{R}^3$ and $p \in X$. We call p a surface point in X if, and only if, there exists a local neighborhood $LN_r(p, X)$ at p with some radius r such that the following conditions hold:*

(i) $\exists n \in \mathbb{N} \exists N_1, N_2, \ldots, N_n \subseteq LN_r(p, X) : LN_r(p, X) = \bigcup_{i=1}^{n} N_i$

(ii) $\forall 1 \leq i < j \leq n : (N_i \cap N_j = \{p\}) \vee (\exists \in \textit{curves} : N_i \cap N_j = l \wedge p \in l)$

(iii) $\forall 1 \leq i \leq n : N_i$ is homeomorphic to D^2

Condition (i) means that an appropriate local neighborhood of p can be covered by a finite number of subsets. The easiest case is that $n = 1$ holds. This means the local neighborhood around p forms a simple surface as indicated in Figures 3-4A and 3-4B. Condition (ii) only allows that any two subsets of the decomposition share either the point p or a curve that contains p. An example for the first case is given in Figure 3-4C by the point p and the two incident grey subsurfaces. An example for the second case is given in Figure 3-4D by the point q and the two incident triangles. Condition (iii) requires that all subsets are topological equivalent to a closed 2D disk. As a result, each surface point has a local neighborhood that does not contain any volume parts, dangling lines, or isolated points but possibly multiple simple surface components.

In general, a surface can be defined as a set that contains only surface points. It ensures that volume parts, dangling lines, or isolated points do not exist in a surface object. Thus, we call a set X a surface set if X contains only surface points. However, it runs into the problem that it may have missing points and lines in the form of punctures and cuts. Thus we define additional concepts to avoid these anomalies. We define the exterior of a set X in R^3. p is an exterior point of X if there exists a neighborhood $N_r(p)$ such that $N_r(p) \cap X = \emptyset$. The set of all exterior points of X is called the $exterior$ of X and is denoted by X^-. We can observe that any missing points or missing lines in a surface set X are not exterior points of the set X. It is because that the neighborhood of a puncture point or a cut line always intersects with X. Thus, we call X a regular surface set if X contains only surface points and $X = \mathbb{R}^3 - X^-$.

we are now able to give the unstructured type definition for complex surfaces in Definition 13.

Definition 13. *The spatial data type surface is defined as*

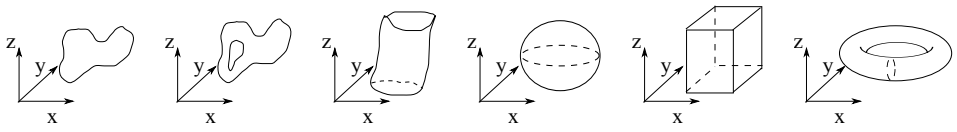


Figure 3-5. Examples of the simple manifold surfaces. Examples of the closed simple manifold surfaces (A,B,C) and examples of the open simple manifold surfaces (D,E,F).

$$\textit{surface} = \{S \subset \mathbb{R}^3 \,|\, (\text{i}) \quad S \text{ is a bounded set}$$
$$(\text{ii}) \quad \forall\, p \in S: \ \text{p is a surface point in } S$$
$$(\text{iii}) \ S = \mathbb{R}^3 - S^-$$
$$(\text{iv}) \ \text{The number of path-connected subsets of } S \text{ is finite}\}$$

We call a value of this type a complex surface.

The structured definition models complex surfaces as objects possibly consisting of several components. Just like separating the point set of a *region2D* object into simple regions, we separate the point set of a *surface* into *simple surfaces*. A *simple surface* object is the non-separable basic unit of a *surface* object. To state a definition of *simple surface*, we start with the definition of simple manifold surfaces in Definition 14.

Definition 14. *The set* sms *of simple manifold surfaces over* $\mathbb{R}^3$ *is defined as*

$$sms = \{S \subset \mathbb{R}^3 \,|\, (\text{i}) \ \ S \text{ is a connected and bounded set}$$
$$(\text{ii}) \ \forall\, p \in S, \exists r \in \mathbb{R}^+, h : LN_r(p, S) \rightarrow D^2,$$
$$\text{where } h \text{ is homeomorphism}\}$$

Condition (i) eliminates disjoint surface sets. Condition (ii) automatically ensures all conditions in the unstructured definition to be satisfied. A manifold surface is, everywhere, locally homomorphic (that is, of comparable structure) to a 2-dimensional disk. Thus it is the simplest form of a surface object. Figure 3-5 shows the examples of simple manifold surfaces. However, simple manifold surfaces describe only a subset of simple surfaces. Condition (ii) is too restrict for simple surfaces in real world and thus can be easily violated. If we pick a corner of a piece of paper and attach it to another point on that paper, the result is not an object of simple manifold surface type.

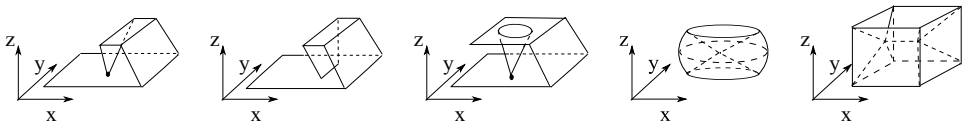


Figure 3-6. Examples of the simple non-manifold surface. Examples of the closed simple non-manifold surfaces (A,B,C) and examples of the open simple non-manifold surfaces (D,E).

Condition (ii) is violated at the attached point, because on that point there does not exist a local neighborhood that is homeomorphic to a 2-dimensional disk. We call such points non-manifold surface points, and we give their definition in Definition 15.

Definition 15. *Let $S \subset \mathbb{R}^3$ and $p \in S$. We call p a non-manifold surface point on S, if there does not exist $r \in \mathbb{R}^+$, so that $LN_r(p, S)$ is homeomorphic to a 2-dimensional disk D^2.*

The Definition 15 indicates that any subset in Euclidean space $\mathbb{R}^3$ containing a non-manifold surface point is not locally homeomorphic to a 2-dimensional disk at that specific non-manifold surface point. With this concept being introduced, we can now give the definition for simple non-manifold surfaces as follows:

Definition 16. *The set* snms *of simple non-manifold surfaces over $\mathbb{R}^3$ is defined as*

$$
\begin{aligned}
snms = \{S \subset \mathbb{R}^3 \,|\, &\text{(i)} \quad S \text{ is a connected and bounded set} \\
&\text{(ii)} \quad \forall\, p \in S, r \in \mathbb{R}^+ : LN_r(p, S) \neq N_r(p) \\
&\text{(iii)} \quad \forall\, p \in S, r \in \mathbb{R}^+ : \exists N \subseteq LN_r(p, S), \\
&\qquad \text{where } N \text{ is homeomorphic to } D^2 \ \} \\
&\text{(iv)} \quad \exists nmp = \{p \in S \,|\, p \text{ is a non-manifold surface point on } S\} : \\
&\qquad S - nmp \text{is a connected set}\}
\end{aligned}
$$

In Definition 16, condition (ii), (iii) excludes volume, tangling lines and tangling points from the definition. Condition(i) eliminates disjoint sets. The simple property and the non-manifold feature are ensured by condition (iv). The existence of non-manifold surface points distinguishes a non-manifold surface object from a manifold surface object. Further, a non-manifold surface object is also simple because the removal of

all non-manifold surface points results in a single connected and non-separable set. Figure 3-6 shows examples of simple non-manifold surfaces. Now the definition for simple surfaces can be given as follows:

Definition 17. *The set* ss *of simple surfaces over* $\mathbb{R}^3$ *is defined as*

$$ss = sms \cup snms$$

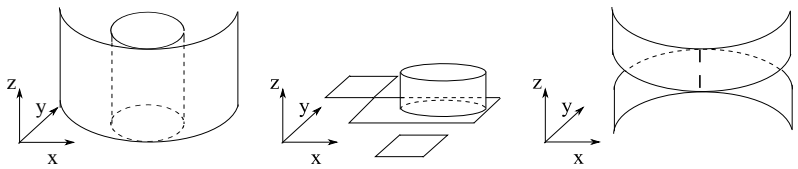
A simple surface therefore is either a simple manifold surface or a simple non-manifold surface. The definition of simple surfaces provides a simplest, non-separable form of a possible surface object. Now, we distinguish the interior, exterior, and the boundary of a surface object. Let $X \subset \mathbb{R}^3$ and $p \in \mathbb{R}^3$, p is an interior point of X if there exists a neighborhood $N_r(p)$ such that $N_r(p) \subseteq X$. The interior of X (X°) is a set that consists of all interior points in X. However, this definition does not work on surface objects, it is because that if $X^\circ \neq \emptyset$ then X is a set with volumes, thus X is not a surface object. As a result, we need new definitions for the interior of a simple surface object. Given a simple manifold surface $S \in ss$, its interior is defined as:

$$\begin{aligned} S^\circ = \{p \in S \mid \exists r \in \mathbb{R}^+ : \\ \text{(i) } \exists h : LN_r(p, S) \to D^2, \text{where } h \text{ is homeomorphism} \\ \text{(ii) } N_r(p) - LN_r(p, S) \text{ is not a connected set}\} \end{aligned}$$

The above definition for the interior of a surface object ensures that the simple surface S is locally homeomorphic to a 2-dimensional disk at its interior points (condition (i)) and the interior points do not lie on the edge of a surface object (condition (ii)). We do not change the definition of the exterior set of a simple surface set X. Thus the boundary of the simple surface X is $\partial X = \mathbb{R}^3 - X^\circ - X^-$, and the closure of the simple surface X is $\overline{X} = X^\circ \cup \partial X$. For example, the point p in Figure 3-4A is a interior point of the surface X, while the point p in Figure 3-4B is a boundary point of the surface X. Moreover, if $\partial S = \emptyset$, we call S an open simple surface because it does not have boundary that encloses its interior (Figure 3-5D-F and Figure 3-6D-E). Otherwise, S is a closed simple surface (Figure 3-5A-C and Figure 3-6A-C).

A more general surface object usually contains a collection of such simple surfaces. For example, Figure 3-7A shows a surface that consists of two connected simple manifold surface objects. A more general surface example is shown in Figure 3-7B, which consists of three connected simple surface objects and a disconnected rectangle. However, it is possible to represent a surface or one of its components by different collections of simple surfaces. Thus, the representation of a general surface object with simple surfaces may not unique. For example, the intersection of two components of a general surface may be a line spanning the interiors of both of them. Then, it is not clear whether we should view the union of the two two components as being two, three, or four simple surfaces. (For an illustration of this, see Figure 3-7C). The intersection may also be a line crossing only part if the interior of one of the two components. In this case, we can view the surface as an intersection of either two or three simple surfaces. Only when the intersection of two such components is a point is there no confusion about what the manifolds are. Thus we need a formal definition of a component to a surface that can only be interpreted one way in so far as the simple surfaces that comprise it. We place certain constraints on the intersection of simple surfaces with other simple surfaces so that we can obtain a definition that provides for uniqueness. Consider therefore the following concept of a composite surface to specify a connected component of a general surface object. This definition incorporates constraints that eliminate ambiguity about the simple surfaces that comprise a general surface.

Definition 18. *The set* cs *of composite surfaces over* $\mathbb{R}^3$ *is defined as*



(A) (B) (C)

Figure 3-7. General surfaces that consist of a collection of simple surfaces.

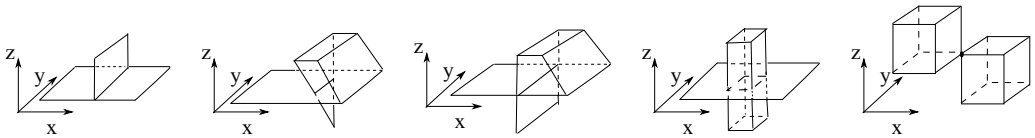


Figure 3-8. Examples of composite surface objects.

$$cs = \{\bigcup_{i=1}^{m} S_i \,|\, \text{(i)}\ \ m \in \mathbb{N}, \forall\, 1 \leq i \leq n : S_i \in ss$$
$$\text{(ii) Let } I = \{p \in \mathbb{R}^3 \,|\, card(S_i \,|\, 1 \leq i \leq m \wedge p \in S_i) > 1\}.$$
$$\text{(a) } \forall\, p \in I : p \text{ is a non-manifold surface point}$$
$$\text{(b) } \forall\, 1 \leq i \leq m : S_i - I \text{ is a connected set}\}$$

In Definition 18, Condition (ii) introduces two constraints on the intersection set I, which contains all points that are shared by more than one simple surfaces. Condition (ii)a expresses that due to the uniqueness constraint a shared point must be a non-manifold point. Otherwise, if there exists a shared point that is not a non-manifold point, then first we can conclude that it is only shared by two simple surfaces. Second, by definition the union of these two simple surface becomes another simple surface because the two simple surfaces will still be connected through that shared point after the removal of non-manifold points. In this case, we can replace the two simple surfaces with their union, and are still be able to represent the same surface objects. This violates the uniqueness constraint. Condition (ii)b expresses that the removal of shared points from every simple surface should not affect their connectivity. Otherwise, the simple surface can be further broken down to smaller simple surface components, and the uniqueness representation constraint will be violated.

The definition of composite surfaces gives a unique interpretation in terms of a collection of simple surfaces for any general connected surface object. Figure 3-8 shows several examples of the composite surface objects. According to the definition, we can identify unique representation for each of these surface objects. For example, Figure 3-8A is a composite surface consists of three rectangles. Figure 3-8B is a

composite surface consists of two simple surfaces, a simple non-manifold and a simple manifold to be specific. Whereas Figure 3-8C shows a composite surface example that consists of three simple surfaces. Further, let CS be a composite surface object with simple surfaces $S_1, \ldots, S_m$, then the interior of CS is $CS^\circ = \bigcup_{i=1}^{m} S_i^\circ$ and the boundary of CS is $\partial CS = \bigcup_{i=1}^{m} \partial S_i$.

Two composite surfaces $c_1, c_2 \in cs$ are disjoint if, and only if, $c_1 \cap c_2 = \emptyset$. The stating of the structured definition for complex surface is now possible. The structured definition of the spatial data type surface is

Definition 19. *The set surface of complex surfaces over* $\mathbb{R}^3$ *is defined as*

$$surface = \{\bigcup_{i=1}^{n} c_i \mid (\mathrm{i}) \;\; n \in \mathbb{N}, \forall\, 1 \leq i \leq n : c_i \in cs$$
$$(\mathrm{ii}) \; \forall\, 1 \leq i < j \leq n : \;\; c_i \text{ and } c_j \text{ are disjoint}\}$$

We call a value of this type complex surface. This definition implies that a complex surface consists of several surface components, which are composite surfaces, and no simple surface in a surface component may intersect with a simple surface in any other surface component. Further, let CMS be a complex surface object with composite surfaces $CS_1, \ldots, CS_m$, then the interior of CMS is $CMS^\circ = \bigcup_{i=1}^{m} CS_i^\circ$ and the boundary of CMS is $\partial CMS = \bigcup_{i=1}^{m} \partial CS_i$.

3.1.2.5 The Data Type *volume*

Similar to the definition of the type $surface$, we also define the data type $volume$ based on the point set topology. Apart from the concepts from point set topology introduced in previous sections, we introduce some additional needed concepts for defining the data type $volume$.

It is obvious that arbitrary 3D point sets do not necessarily form a volume. But open and closed point sets in $\mathbb{R}^3$ are also inadequate models for volumes since they can suffer from undesired geometric anomalies. A volume defined as an open point set runs into the problem that it may have missing points, lines, and surfaces in the form of

punctures, cuts, and stripes. At any rate, its boundary is missing. A volume defined as a closed point set admits isolated or dangling point, line, and surface features.

Regular closed point sets [101] avoid these anomalies. Let $X \subset \mathbb{R}^3$. X is called regular closed if, and only if, $X = \overline{X^\circ}$. The effect of the interior operation is to eliminate dangling points, dangling lines, dangling surfaces, and boundary parts. The effect of the closure operation is to eliminate punctures, cuts, and stripes by appropriately supplementing points and adding the boundary. Due to their definition, closed neighborhoods are regular closed sets.

With the definition of bounded and connected sets introduced, we are now able to give an unstructured type definition for volumes in Definition 20.

Definition 20. *The spatial data type volume is defined as*

$$\begin{aligned} \textit{volume} = \{V \subset \mathbb{R}^3 \mid &\text{(i)} \quad V \text{ is regular closed} \\ &\text{(ii)} \quad V \text{ is bounded} \\ &\text{(iii) The number of connected sets of } V \text{ is finite}\} \end{aligned}$$

We call a value of this type a complex volume.

Although this definition tells us what a volume is, it does not reveal how one builds a volume. As for the other data types, therefore, we need to construct a structured definition of *volume* which enforces the uniqueness constraint about the components which comprise a *volume*. The structured definition models complex volumes as objects possibly consisting of several components and possibly having cavities. It distinguishes simple volumes, simple volumes with cavities(also called solids), and complex volumes. In order to obtain a more fine-grained and structured view of volumes, the simple volumes are further distinguished with manifolds and simple non-manifolds, we start with the definition of the simplest volume object, which we call the manifolds, in Definition 21.

Definition 21. *The set* manifolds *over* $\mathbb{R}^3$ *is defined as*

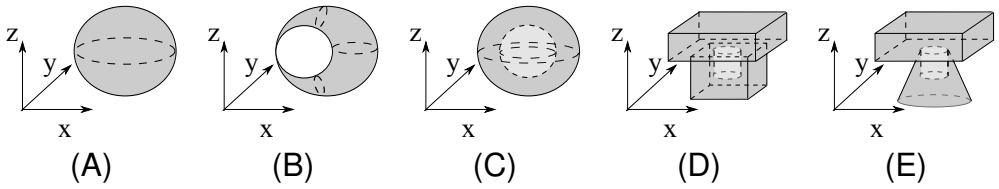


Figure 3-9. Simple volume examples. Examples of a manifold (A), a non-manifold (B), and simple volumes with cavities (C,D,E).

$$
\begin{aligned}
\mathit{manifolds} = \{V \subset \mathbb{R}^3 \,|\, &\text{(i)} \quad V \text{ is regular closed} \\
&\text{(ii)} \quad V \text{ is bounded} \\
&\text{(iii)} \quad \mathbb{R}^3 - V \text{ is a connected set} \\
&\text{(iV)} \quad \forall\, p \in V, \exists r \in \mathbb{R}^+, h : LN_r(p, V) \to D^3, \\
&\qquad \text{where } h \text{ is homeomorphism}\}
\end{aligned}
$$

This definition gives the formal description of a simplest volume object in 3D space, that has connected interior without anomalies (condition (i) and condition (ii)), has no cavities (condition (iii)), and is everywhere locally homomorphic, i.e. topologically equivalent, to a three-dimensional ball (condition (iv)). Figure 3-9A shows an example of a manifold. However, this describes only a subset of simple volumes. The condition (iv) is too strict for a large number of other simple volume objects. There exist another set of simple volumes, which we call the simple non-manifold volumes. To define the simple non-manifolds, we first refine the definition in Definition 15 and define the concept of non-manifold volume points in Definition 22.

Definition 22. *Let $V \subset \mathbb{R}^3$ and $p \in S$. We call p a non-manifold volume point on V, if there does not exist $r \in \mathbb{R}^+$, so that $LN_r(p, V)$ is homeomorphic to a 3-dimensional ball D^3.*

The Definition 22 indicates that any point set in $\mathbb{R}^3$ Euclidean space containing a non-manifold volume point is not locally homeomorphic to a 3-dimensional ball at that specific non-manifold volume point. With this concept being introduced, we can now give the definition for simple non-manifolds as follows:

Definition 23. *The set* snm *of simple non-manifolds over $\mathbb{R}^3$ is defined as*

$$
\begin{aligned}
snm = \{ V \subset \mathbb{R}^3 \,|\, &\text{(i)} \quad V \text{ is regular closed} \\
&\text{(ii)} \quad V \text{ is bounded} \\
&\text{(iii)} \ \mathbb{R}^3 - V \text{ is a connected set} \\
&\text{(iv)} \ \exists nmp = \{ p \in V \,|\, \text{p is a non-manifold volume point} \} : \\
&\qquad V - nmp \text{is a connected set } \}
\end{aligned}
$$

In general, volumes described in Definition 23 are volume objects that have connected interior without anomalies (condition (i) and condition (ii)) and have no cavities (condition (iii)). But they involve non-manifold volume points (condition (iv)), at which is not locally homeomorphic to a 3-dimensional ball. This feature distinguishes them from manifolds we defined in Definition 21. Further, the condition (iv) ensures it to be simple because the removal of all non-manifold volume points results in a single connected and non-separable set. Figure 3-9B shows an example of a simple non-manifold. Now the definition for simple volumes can be given as follows:

Definition 24. *The set* sv *of simple volumes over* $\mathbb{R}^3$ *is defined as*

$$sv = manifolds \cup snm$$

This, in particular, means that a simple volume is either a manifold or a simple non-manifold that has a connected interior, a connected boundary, and a single connected exterior. Hence, it does not consist of several components, and it does not have cavities. The concept of a cavity is topologically not directly inferable since point set topology does not distinguish between “outer” exterior and “inner” exterior of a set. This requires an explicit and constructive definition of a volume containing cavities and the embedding of a volume within a volume by use of set operations. Thus the definition for simple volumes with cavities can be given as follows:

Definition 25. *The set* svc *of simple volumes with cavities, also called solids, over* $\mathbb{R}^3$ *is defined as*

$$
\begin{aligned}
svc = \{V \subset \mathbb{R}^3 \mid (\mathrm{i}) \quad & \exists V_0, \ldots, V_n \in sv : V = V_0 - \bigcup_{i=1}^{n} V_i^\circ \\
(\mathrm{ii}) \quad & V \text{ is connected} \\
(\mathrm{iii}) \; & \forall\, 1 \leq i \leq n : V_i \subset V_0 \wedge \\
& (\partial V_0 \cap \partial V_i = \emptyset \vee (\partial V_0 \cap \partial V_i) \in (point3D \cup line3D)) \\
(\mathrm{iv}) \; & \forall\, 1 \leq j < k \leq n : V_j \cap V_k = \emptyset \vee (V_j \cap V_k) \in (point3D \cup line3D)\}
\end{aligned}
$$

For $V \in svc$ with $V = V_0 - \bigcup_{i=1}^{n} V_i^\circ$, $V_0, \ldots, V_n$ are called cavities. Condition (i) determines the point set of a simple volume with cavities. Condition (ii) requires that despite the subtraction of point sets in (i) the remaining point set remains connected. Condition (iii) and (iv) allow a cavity within a solid to touch the boundary of V_0 or of another cavity either in 3D point objects or 3D line objects but not in common, partial surfaces. This is necessary to ensure uniqueness of representation and to achieve closure under the geometric operations union, intersection, and difference. For example, subtracting a solid A from a solid B may lead to such a cavity in B. On the other hand, to allow two cavities to have a partially common surface makes no sense because then adjacent cavities could be merged to a single cavity by eliminating the common surface. Figure 3-9C and Figure 3-9D show examples of simple volumes with cavities.

Let $V = V_0 - \bigcup_{i=1}^{n} V_i^\circ$ be a simple volume with cavities $V_0, \ldots, V_n$. Then the boundary of V is given as $\partial V = \bigcup_{i=0}^{n} \partial V_i$ and the interior of V is given as $V^\circ = V_0^\circ - \bigcup_{i=1}^{n} V_i$. Now, what remains is to extend the definition of *svc* so that we have a data type *volume* that has multi-component objects. The structured definition of *volume* is given in Definition 26

Definition 26. *The set* volume *of complex volumes over* $\mathbb{R}^3$ *is defined as*

$$
\begin{aligned}
volume = \{V \subset \mathbb{R}^3 \mid (\mathrm{i}) \quad & \exists V_1, \ldots, V_n \in svc : V = \bigcup_{i=1}^{n} V_i^\circ \\
(\mathrm{ii}) \; & \forall\, 1 \leq j < k \leq n : V_j \cap V_k = \emptyset \vee (V_j \cap V_k \in point3D) \vee \\
& ((V_j \cap V_k \in line3D) \wedge \\
& (\neg \exists B \in blocks : \mathrm{B \text{ is a closed block component in } } V_j \cap V_k))\}
\end{aligned}
$$

We call a value of this type complex volume. The definition requires of a solid to be disjoint from another solid, or to meet another solid in one or several single boundary

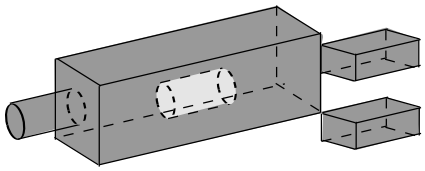


Figure 3-10. An example volume. The volume has three (and not four) components. The largest component contains a cavity. We could also say that this picture has three (and not four) volumes.

points or curves, or to lie within a cavity of another solid and possibly share one or several single boundary points or curves with the boundary of the cavity. Solids having common surface parts with other solids or cavities are disallowed. The argumentation is similar to that for the solid definition. However, here we have an additional constraint that requires the intersection of two solids to be a line object that contains no closed curve components. It is necessary to achieve the uniqueness of representation. Without this constraint, we can not ensure that the meet of two solids will not create cavities. Consider the following scenario in Figure 3-9E, two solids meet at a ring, and as a result, a cavity is created. Thus without the constraint in condition (ii), it is difficult to say if the volume in Figure 3-9E is a single component simple volume with cavities or is a two component complex volume with no cavities. Our definition can avoid this and uniquely identify this volume as a simple volume with one cavity. An example of a complex volume object is shown in Figure 3-10, where the volume consists of three (and not four) components.

Let $V = \bigcup_{i=1}^{n} V_i$ be a volume with $V_1, \ldots, V_n \in svc$. Then the boundary of V is $\partial V = \bigcup_{i=1}^{n} \partial V_i$ and the interior of V is $V^\circ = \bigcup_{i=1}^{n} V_i^\circ = V - \partial V$. The closure of V is $\overline{V} = \partial V \cup V^\circ$, and the exterior of V is $V^- = \mathbb{R}^3 - \overline{V}$.

3.1.3 Three-dimensional Operations

Section 3.1.3.1 gives an overview for all operations belonging to the first two categories. We give the signature and the semantics of operations in the first category,

Table 3-1. Sets and enumerations of 3D spatial data types.

Type set name	Set of spatial data types	Enumeration of spatial data types
SPATIAL 3D	$\{point3D, line3D, surface, volume\}$	$\{\alpha_0, \alpha_1, \alpha_2, \alpha_3\}$

which yield another spatial object, in Section 3.1.3.2. The operations in the second category that yield numerical values are introduced in Section 3.1.3.3.

3.1.3.1 Overview

This section presents a summary of the operations for 3D spatial data types and a classification of them by categories such as geometric operations, predicates, etc. It also will explain the notation used for the signatures of the operations. Finally, it presents a sample database upon which sample queries will illustrate the operations. Operations in the algebra *SPAL3D* are very important as they form the basis for many operations in queries involving spatial objects. There are three principles in the design of the operations in SPAL3D. First of all, the operations must be as global as possible. Secondly, they must determine the interesting features of spatial objects. Finally, the specifications of the operations contain both their signatures and semantics. The signatures of the operations show that most of the operations are polymorphic operations, in that there is overloading on many of their inputs and their outputs. The first principle prevents a proliferation of operations. Global operations result in the avoiding of this proliferation by overloading operations so that as many of the spatial data types as possible can perform the same operation. Thus, there is usually only one definition for each operation even when SPAL3D permits the operation to have several or all of the spatial data types as operands. For this purpose, introduced in Table 3-1 is the type set SPATIAL3D as well as an enumeration over it.

Thus, $\alpha_0 = point3D$, $\alpha_1 = line3D$, $\alpha_2 = surface$ and $\alpha_3 = volume$. It is by using the enumeration that we overload the operations and simplify the notation defining them.

The second principle deals with the need of users for operations that produce results with different types. Because of user requirements, we must have these kind

Table 3-2. Classification of 3D operations.

Category	Sub-category	Operations
Geometric operations	Geometric set operations	*union*,*difference*,*symmetricdifference*, *intersection*,$common_0$,$common_1$,$common_2$
Geometric operations	Affine transformations	*translate*,*scale*,*rotate*,*revolve*,*reflectyz*, *reflectxy*,*reflectxz*
Geometric operations	Other geometric operations	$convexhull_1$,$convexhull_2$,$convexhull_3$,*projectxy*, *projectxz*,*projectyz*,*boundingbox*,*boundary*
Numerical operations		*length*,*perimeter*,*area*,*surfacearea*,*volume*, *xmaximum*,*ymaximum*,*zmaximum*,*xminimum*, *yminimum*,*zminimum*,*distance*,*cardinality*,
Spatial predicates	Topological predicates	*disjoint*,*meets*,*covers*,*coveredby*,*inside*, *contains*,*overlaps*,*equals*
Spatial predicates	Directional predicates	*north*,*south*,*east*,*west*,*nothwest*,*northeast*, *southwest*,*southeast*,*samelocation*

of operations. The most elegant operations yield either another spatial object (i.e., geometric set operations, affine transformations, and other geometric operations) as their result or a boolean value (i.e., predicates). The operations that are algebraically closed are especially elegant. More mundane operations, such as those that determine an area or an average for spatial objects, are also important, though closure issues for them are irrelevant. This is because users view them as important. These more mundane operations are numerical operations. Table 3-2 summarizes the operations about to be defined by listing their names and the class of operations into which they fall.

Thirdly, when defining operations, there are both signatures and semantics. The semantics of an operation, of course, are formulas that produce the result of the operation. A signature $\gamma \times \delta \rightarrow \xi$ of a binary operation means that the operands in the operation are type γ and δ and the result is type ξ. Though the two operands may be different types, however, they must have values that fall within the same space. If the two operands are the same type, the signature is $\alpha \times \alpha \rightarrow \xi$. When the two operands are the same type for an operation that yields another spatial object, then the result might be the same type as the operands. This is closure of an operation. So, for some geometric

set operations, it is true that $\alpha \times \alpha \rightarrow \alpha$. Also, we will always use *A* and *B* as notation to represent an operand of any spatial data type.

Finally, while defining the operations we illustrate their usefulness by formulating sample queries in a hypothetical, but plausible extension of SQL. The queries indicate how the 3D data types can exploit database support. Because the spatial data types are abstract, they can appear in relational schemas as attributes types like the other standard data types. We will use the following example databases for the sample queries.

DB1: An astronomical database dealing with solar systems and containing information on the stars, planets, and asteroids present within the solar systems.

```
solarsystem(name: string, diameter: integer, place: volume)
star(name: string, diameter: integer, center: point3D)
planet(name: string, diameter: integer, orbit: line3D)
asteroid(name: string, diameter: integer, orbit: line3D)
```

DB2: A Geographic Information System (GIS) database for mountains, river system, soil system, buildings and pipelines.

```
mountain(name: string, geoShape: volume)
river_system(name: string, type: string, area: surface)
soil(texture: string, geoShape: volume)
building(name: string, year:integer, shape: volume)
pipeline(id: integer, type: string, shape: line3D)
```

3.1.3.2 Geometric Operations

Geometric operations are the operations that produce spatial objects as their result. The operations that result in another spatial object are those that most distinguish the operations of spatial database systems from database systems that do not have a spatial component. This makes them of a different nature than the operations in non-spatial database systems. From the perspective of functionality, we can further distinguish the geometric operations to three sub-categories: the geometric set operations, the affine transformations, and the other geometric operations.

- Geometric set operations

Geometric set operations perform unions, differences, symmetric differences, and intersections between two spatial objects. Geometric set operations can have as operands spatial objects of all spatial data types. An desired property for geometric set operations is the closure property, that is operations produce a result with the same data type as the operands. For example, it is nice to have the intersection of two volumes produces a volume. However, this is not guaranteed in the real world scenario, and it is mathematically natural that geometric set operations sometimes produce a result that has pieces with a lower dimension than the operands. For example, the intersection of two surfaces often produces a line3D object, and it may even produce a combination of line3D objects and surface objects. More importantly, users may be interested in the lower-dimensional pieces of an operation.

This is not a problem for union, difference, and symmetric difference operations because they always produce a result that has a data type equal to the operands when the operands have the same type. In another word, these three operations are closed. Intersection operations, however, often produce a mixture of point sets of different spatial data types, thus are not closed in practical. Thus, we design two different kinds of intersection operations. One kind is the theoretically elegant operation that has closure, and the other kind captures pieces of intersection operations with different dimensionality.

Thus, we first define the kind of intersection operations that returns lower-dimensional pieces of the intersection between two operands. Since each operation can only produce results of one data type, and we have spatial data types for three different lower-dimensional spatial objects-0D,1D, and 2D, there are three such operations. We call them the $common_k$ operations whree k is equal to the dimension of the resultant spatial object. Each $common_k$ operation produces the pieces of an intersection equal to a spatial object of a single dimension with all other pieces thrown away. Thus, $common_0$ operation produces only a $point3D$ object, the $common_1$ operation produces only a

Operations	Signature	Semantics
$common_0$	$\alpha_i \times \alpha_j \rightarrow \alpha_k$ where $0 \leq k < i \leq 3$ $0 \leq k < j \leq 3$	$\{p \in A \cap B \mid p$ *is isolated in* $A \cap B\}$ if $A \in line3D, B \in \alpha_i, 1 \leq i \leq 3$ $\{p \in \partial A \cap \partial B \mid p$ *is isolated in* $\partial A \cap \partial B\}$ if $A, B \in \alpha_i, 2 \leq i \leq 3)$
$common_1$		$\bigcup\{I \subset (\partial A \cap \partial B) \cup (\partial A \cap B^\circ) \cup (A^\circ \cap \partial B) \cup (A^\circ \cap B^\circ) \mid I \in line3D\}$ if $A, B \in surface$ $\bigcup\{I \subset (\partial A \cap \partial B) \cup (A^\circ \cap \partial B) \mid I \in line3D\}$ if $A \in surface, B \in volume$ $\bigcup\{I \subset \partial A \cap \partial B \mid I \in line3D\}$ if $A, B \in volume$
$common_2$		$\bigcup\{s \subset (\partial A \cap \partial B)^\circ \mid s \in surface\}$ if $A, B \in volume$

Figure 3-11. These operations are intersection operations that yield a result of lower dimension than either operand.

line3D object, and the $common_2$ produces only a *surface* object. Figure 3-11 gives definitions for the $common_k$ operations.

With the unclosed geometric set operations being defined separately, we can now define four closed geometric set operations *union*, *difference*, *symmentricdifference*, and *intersection*, that only produce a spatial object of the same type as its operands if the operands have the same spatial type. In the case when two operands have different spatial types, they produce a result that has the same data type as at least one of the operands. The result of a *union* operation is the regularized set union of the two operands. It returns the higher-dimensional operand when its operands have different spatial types. Thus the result of a *union* operation between objects of different types is not interesting. Syntactically, however, it is meaningful to allow *union* operations for operands of different data types because operations may be nested. Then, failure to allow such an operation could cause the *union* operation to be undefined.

The result of the *difference* operation is the regularized set difference of the first object minus the second object. Its result is the same type as the first operand, but not necessarily the same value as the first operand. The operand types may be any

combination of types in SPATIAL3D, even though some of them are not relevant. Only those combinations of operand types return new results when the dimension of the second operand is equal or higher than the dimension of the first operand. If the dimension of the second operand is smaller, then the first operand is returned unchanged. The *difference* operation must therefore apply closure to the result to avoid the generation of anomalies.

The signature of the *symmetricdifference* operation is the same as the *union* operation. It also discards all lower-dimensional parts. It also has an uninteresting, but syntactically necessary result when the operands are different types. As with the *difference* operation, however, we must apply closure to the result to avoid anomalies.

The *intersection* operation yields a result that is of dimension equal to the operand of lower dimension. As previously noted, a set intersection may theoretically include pieces of a dimension lower than the operand of lower dimension. But since we have defined three other kind of intersection operations for handling such pieces, in this *intersection* operation, we discard all pieces of the point set other than the highest-dimensional parts.

Figure 3-12 gives signatures and semantics for the geometric set operations that are closed. Now, we use some examples to illustrate queries for the geometric set operations in a hypothetical extension of SQL. In Q1, we present a query requiring one of the $common_k$ operations based on the solar system database previously specified. Q2 and Q3 demonstrate queries for the closed geometric set operations using the GIS database previously specified.

Q1: Find places where there could be a collision between Earth and an asteroid.

```
SELECT common[0](planet.orbit,asteroid.orbit)
FROM planet, asteroid
WHERE planet.name = 'Earth'
```

Q2: Find the part of sewer pipelines that belongs to the Physics building .

Operations	Signature	Semantics
union	$\alpha \times \alpha \rightarrow \alpha$	$A \cup B$
	$\alpha_i \times \alpha_j \rightarrow \alpha_i \mid$ $\alpha_j \times \alpha_i \rightarrow \alpha_i$ where $0 \leq j < i \leq 3$	$A \mid B$
difference	$\alpha_i \times \alpha_j \rightarrow \alpha_i \mid$ where $0 \leq i, j \leq 3$	$\overline{A - B}$
symmetric difference	$\alpha_i \times \alpha_j \rightarrow \alpha_i \mid$ $\alpha_j \times \alpha_i \rightarrow \alpha_i$ where $0 \leq j < i \leq 3$	$\overline{(A - B) \cup (B - A)}$
intersection		$A \cap B$ if $A \in$ *point3D*, $B \in \alpha_i, 0 \leq i \leq 3$ $(A \cap B) - common_0(A, B)$ if $A \in$ *line3D*, $B \in \alpha_i, 1 \leq i \leq 3$ $(A \cap B) - (common_0(A, B) \cup common_1(A, B))$ if $A \in$ *surface*, $B \in \alpha_i, 2 \leq i \leq 3$ $\overline{(A \cap B)^\circ}$ if $A, B \in$ *volume*

Figure 3-12. These geometric set operations yield a result that has the same spatial data type as one or both of the operands.

```
SELECT intersetion(pipeline.shape, building.shape)
FROM pipeline, building
WHERE pipline.type='sewer' AND building.name='Physics'
```

Q3: Find the soil with texture type as sandy clay that is not part of the Great Smoky Mountains.

```
SELECT difference(soil.geoShape, mountain.geoShape)
FROM soil, mountain
WHERE soil.texture='sandy clay'
      AND mountain.name='Great Smoky Mountains'
```

- Affine transformations

Affine transformations are operations that change a currently existing spatial object without changing its topology. They move, enlarge or reduce in size of a spatial object. They may produce an object that mirrors the original object. Thus, as with geometric set operations, affine transformations result in a spatial object. However, the word affine

implies that the spatial object that results has the same topology as the original object. With the possible exception of the reflect operations, the resultant object has the exact same shape. It is just shifted to another location or has changed its size. The affine transformations are the operations *translate*, *scale*, *rotate*, and the *reflect* operations. Some types of rotation can also be called *revolve*. The three reflect operations are *reflectxy*, *reflectyz*, and *reflectxz*.

The *translate* operation is relatively easy. It simply shifts the location of spatial objects to a different place in 3D space. Thus, in addition to the spatial object that the operation translates, its operands involve the x, y, and z-values that give the amount and direction of the translation. More specifically, we add to the coordinates of each point in the translated object the size of the translation in the x, y, and z directions respectively. The x, y, and z-values by which an object is translated amount to a 3D point. Thus, the signature of the *translate* operation includes the *point3D* set. We may also view the *translate* operation as moving the location of the origin of the axes rather than moving the object itself, but only for the translated object.

Scaling has to do with increasing or decreasing the size of a spatial object. When scaling, we increase or decrease the object's size by a specific factor in each direction with respect to a reference point. The reference point is a point at the center of the original object, which remains to be the center of the resultant object after scaling. Scaling with respect to the center of an object ensures that the resultant object has the same topology as the original object. The reference point used for the scaling here is the center of the minimum spanning sphere of the scaled object. The minimum spanning sphere of a geometric object is also called the bounding ball which is the smallest sphere containing the geometric object. This applies to all types of objects, e.g. *volume*, *point3D*.

Rotation turns an object around a reference point. The rotation formulas in this article specify rotation with respect to the reference point that is at the center of an

object. A spatial object may rotate about its own center or it may be in rotation about another object. The rotation of the object itself is about the center of the object's own minimum spanning sphere. For rotation of an object about another object, the rotation is about the center of the other object's minimum spanning sphere This second type of rotation is really revolve. In either case, whether we rotate the spatial object itself or revolve the object about another object, we must determine the point at the center of the minimum spanning sphere of the object about which the rotation takes place. The name of the operation that rotates a spatial object about its own center is *rotate* and the name of the operation that rotates an object about another object is *revolve*. For both of these operations, the shape remains the same.

The *rotate* operation result in an angular movement of the spatial object relative to the object's center so that the distance of each point in the object from the center of the minimum spanning sphere remains fixed. The angular movement of each point of the object can be decomposed into an angular movement θ in the xy plane and an angular movement ϕ relative to the z-axis. In effect, these rotation definitions transform the coordinates of the object's point set to spherical coordinates, add the angles from the operands of the operation to the two angles within the spherical coordinates of each point and transform the result back to rectangular coordinates. The author believes that these definitions are better for users of a spatial database system because it frees users from having to specify the axis of the rotation. They do have to specify angular movement, but the author believes that this is easier than specifying an axis of rotation.

The definitions for *rotate* and *revolve* are almost the same, but their signatures and semantics are both slightly different. Both signatures include the two angles of rotation and the rotated object. The *revolve* operation, in addition, has an operand for the object about which the rotation takes place. The only difference in the semantics is that, for *rotate*, the center of the rotation is the rotating object's own minimum spanning sphere

Operations	Signature	Semantics
translate	$\alpha \times point3D \rightarrow \alpha$	$\{(x + x_0, y + y_0, z + z_0) \mid (x, y, z) \in A \wedge x_0, y_0, z_0 \in \mathbb{R}\}$
scale		$\{(x + a(x - x_c), y + b(y - y_c), z + c(z - z_c)) \mid (x, y, z) \in A \wedge a, b, c \in \mathbb{R}\}$ where (x_c, y_c, z_c) is the point at the center of the minimum spanning sphere of A
rotate	$\alpha \times real \times real \times real \rightarrow \alpha$	$\{(x_c + \rho \sin(\phi + \arccos \frac{z - z_c}{\rho}) \cos(\theta + \arctan \frac{y - y_c}{x - x_c}), y_c + \rho \sin(\phi + \arccos \frac{z - z_c}{\rho}) \sin(\theta + \arctan \frac{y - y_c}{x - x_c}), z_c + \rho \cos(\phi + \arccos \frac{z - z_c}{\rho})) \mid (x, y, z) \in A, \rho = \sqrt{(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2}\}$ where θ and ϕ are the angles of rotation of A in the xy plane and relative to the z-axis, and (x_c, y_c, z_c) is the point at the center of the minimum spanning sphere of A
revolve	$\alpha_i \times \alpha_j \times real \times real \rightarrow \alpha$ where $0 \leq i, j \leq 3$	$\{(x_c + \rho \sin(\phi + \arccos \frac{z - z_c}{\rho}) \cos(\theta + \arctan \frac{y - y_c}{x - x_c}), y_c + \rho \sin(\phi + \arccos \frac{z - z_c}{\rho}) \sin(\theta + \arctan \frac{y - y_c}{x - x_c}), z_c + \rho \cos(\phi + \arccos \frac{z - z_c}{\rho})) \mid A \in \alpha_i, (x, y, z) \in A, \rho = \sqrt{(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2}\}, B \in \alpha_j$ where θ and ϕ are the angles of rotation of A in the xy plane and relative to the z-axis, and (x_c, y_c, z_c) is the point at the center of the minimum spanning sphere of B
reflectxy	$\alpha \rightarrow \alpha$	$\{(x, y, -z) \mid (x, y, z) \in A\}$
reflectyz		$\{(-x, y, z) \mid (x, y, z) \in A\}$
reflectxz		$\{(x, -y, z) \mid (x, y, z) \in A\}$

Figure 3-13. Affine transformations.

while, for *revolve*, the center is the minimum spanning sphere of the object about which the other object revolves.

Reflection is producing the mirror image of an object. We can reflect an object through any of the three axes planes as if the planes were mirrors. The *reflectxy*, *reflectyz*, and *reflectxz* operations yield a spatial object after it has been reflected through the xy-plane, yz-plane, and xz-plane respectively. Mathematically the reflect operations are simple because they only require negating one of the coordinate values. To reflect a spatial object, we negate its z coordinate values for *reflectxy*, its x coordinate values for *reflectyz* and its y coordinate values for *reflectxz*.

Figure 3-13 gives signatures and semantics for the affine transformations. An example (Q4) is used to illustrate queries for the affine transformations.

Q4: Show what the Physics building looks like if it is rotated 30 degrees counter-clockwise and shifted 10 meters to the east.

```
SELECT translate(rotate(building.shape, 30, 0), 10, 0, 0)
FROM building
WHERE building.name='Physics'
```

- Other geometric operations

Additional spatial operations are the ones other than geometric operations and affine transformations that produce a spatial object as their result. The result of the operations is, generally, a part or an approximation of the spatial object upon which the operation is applied. Often, these operations have only one operand. Some of them apply to only one or a subset of the 3D spatial data types. The spatial operations are the convex hull operations, the projection operations, boundingbox, and boundary.

The convex hull of a spatial object is the minimum convex object that contains the object. In Euclidean space, an object is convex if for every pair of points within the object, the straight segment that connects them is entirely within the object. For example, a solid cube is convex, but anything that is hollow or has a dent in it, for example, a crescent shape, is not convex. Thus, the convex hull of a spatial object is the smallest convex object that contains the object. In another word, a spatial object CH is a convex hull of the spatial object X, if and only if CH is convex and $X \subseteq CH$, and for any convex object C such that $X \subseteq C$, we have $CH \subseteq C$. In general, the convex hull of a spatial object of any 3D spatial type, i.e., *point3D*, *line3D*, *surface* or *volume*, is a convex spatial object of type *volume* according to the definition. But similar to the *intersection* operation, anomalies also occur for the convex hull operation. For example, when computing convex hull for a *line3D* object whose points are collinear, or a *surface* object whose points are coplanar, or a *point3D* object whose points are either collinear or coplanar, the convex hull is no more a object of type *volume*, but a 3D segment (type

line3D) or a convex 3D polygon (type *surface*). Thus, we must regularize the result of a convex hull operation by asking the user to specify the desired dimensionality of a convex hull. Since we can only have any convex hull as a 1D object (*line3D*), a 2D object (*surface*), or a 3D object (*volume*), we define three convex hull operations with the name $convexhull_k$, where k is equal to the dimension of the resultant spatial object and $1 \leq k \leq 3$. Each operation produces a convex spatial object with a specific dimensionality. Figure 3-14 contains the signature and the semantics of the three convex hull operations.

The projection operations produce an approximation of a spatial object. The approximation is a mapping of the object into one of the axes planes. The *projectxy* operation projects an object into the xy-plane, the *projectyz* operation projects it into the yz-plane, and the *projectxz* operation produces a projection into the xz-plane. The semantics of the projection operations are simple. We merely set to zero the appropriate coordinate for all points in the projected spatial object. The signatures for projections indicate that the resultant object exists in a lower-dimensional space that the original object. In fact, they indicate that, except for *volumes*, projections produce a two-dimensional (2D) version of the projected object in 3D if we eliminate the zero-ed out coordinate. Thus, projections of *point3D* objects produce *point2D* objects, *line3D* objects produce *line2D* objects, and *surfaces* produce *region2D* objects. Projections of *volumes* also produce *region2D* objects.

The anomalies that occur for the *intersection* operation also occur for projection operations. It is possible that part or all of a 3D object is perpendicular to the plane into which the object is projected. (Note, however, that it is not possible for part of a *volumes* to be perpendicular to a plane.) In such a case, the part of the object that is perpendicular to the plane will project into an object that has parts of differing dimensions. Thus, as we did for the *intersection* operations, we must regularize

the result of projection operations by throwing out the lower-dimensional parts. The signatures of the projection operations indicate this and the semantics reflect it.

The *boundingbox* operation results in a box that encloses an entire spatial object even when the object is a multi-component object. Thus, it is different from the other spatial operations in this subsection because its result is bigger than its operand. The result is, however, a crude approximation of its operand. As its signature indicates, we can find the *boundingbox* of any type of spatial object, even a complex point object, but the result is always a volume.

When we apply the *boundary* operation to an object, the result is its boundary. Therefore, to understand the result of this operation, remember that the boundaries of *volumes* are *surfaces*, of *surfaces* are lines, and of lines are points. Points have no boundaries and, therefore, we cannot apply the *boundary* operation to *point3D* objects.

Figure 3-14 gives signatures and semantics for the other spatial operations. Recall the distance function d. Define the following 2D closed neighborhoods of radius ϵ: $N_{\epsilon_{xy}}(x_1, y_1, 0) = \{(x, y, 0) | d((x, y, 0), (x_1, y_1, 0)) < \epsilon\}$, $N_{\epsilon_{yz}}(0, y_1, z_1) = \{(0, y, z) | d((0, y, z), (0, y_1, z_1)) < \epsilon\}$, and $N_{\epsilon_{xz}}(x_1, 0, z_1) = \{(x, 0, z) | d((x, 0, z), (x_1, 0, z_1)) < \epsilon\}$. Recall also that α is a type variable ranging over SPATIAL3D.

The following examples (Q5, Q6 and Q7) illustrate queries for other geometric operations.

Q5: Find convex hull of Earth's solar system.

```
SELECT convexhull[3](solarsystem.place) FROM  solarsystem
WHERE solarsystem.name='Solsystem'
```

Q6: Extract the wall of the Physics building.

```
SELECT boundary(building.shape) FROM building
WHERE building.name = 'Physics'
```

Q7: Get the base map of the Chemistry building.

Operations	Signature	Semantics
$convexhull_1$	$\alpha \rightarrow line3D$	$\{L \in line3D \mid \forall S \subseteq \mathbb{R}^3, A \subseteq S, S$ is convex : $A \subseteq L \wedge L \subseteq S\}$
$convexhull_2$	$\alpha \rightarrow surface$	$\{L \in surface \mid \forall S \subseteq \mathbb{R}^3, A \subseteq S, S$ is convex : $A \subseteq L \wedge L \subseteq S\}$
$convexhull_3$	$\alpha \rightarrow volume$	$\{L \in volume \mid \forall S \subseteq \mathbb{R}^3, A \subseteq S, S$ is convex : $A \subseteq L \wedge L \subseteq S\}$
projectxy	*point3D* $\rightarrow$ *point2D*	S_{xy} if $A \in point3D \vee A \in volume$ $S_{xy} - \{p \in S_{xy} \mid p$ is isolated in $S_{xy}\}$ if $A \in line3D$ $S_{xy} - \{p \in S_{xy} \mid \forall \epsilon, N_{\epsilon_{xy}} \nsubseteq S_{xy}\}$ if $A \in surface$ where $S_{xy} = \{(x, y, 0) \mid (x, y, z) \in A\}$
projectyz	*line3D* $\rightarrow$ *line2D* *surface* $\rightarrow$ *region2D* *volume* $\rightarrow$ *region2D*	S_{yz} if $A \in point3D \vee A \in volume$ $S_{yz} - \{p \in S_{yz} \mid p$ is isolated in $S_{yz}\}$ if $A \in line3D$ $S_{yz} - \{p \in S_{yz} \mid \forall \epsilon, N_{\epsilon_{yz}} \nsubseteq S_{yz}\}$ if $A \in surface$ where $S_{yz} = \{(0, y, z) \mid (x, y, z) \in A\}$
projectxz		S_{xz} if $A \in point3D \vee A \in volume$ $S_{xz} - \{p \in S_{xz} \mid p$ is isolated in $S_{xz}\}$ if $A \in line3D$ $S_{xz} - \{p \in S_{xz} \mid \forall \epsilon, N_{\epsilon_{xz}} \nsubseteq S_{xz}\}$ if $A \in surface$ where $S_{xz} = \{(x, 0, z) \mid (x, y, z) \in A\}$
bounding box	$\alpha \rightarrow volume$	$\{(x, y, z) \in \mathbb{R}^3 \mid$ $xminimum(A) \leq x \leq xmaximum(A) \wedge$ $yminimum(A) \leq y \leq ymaximum(A) \wedge$ $zminimum(A) \leq z \leq zmaximum(A)\}$
boundary	$\alpha_i \rightarrow \alpha_{\lfloor i/2 \rfloor}$ $(1 \leq i \leq 3)$	∂A

Figure 3-14. Other geometric operations. Figure 3-15 has definitions of xminimum, xmaximum, yminimum, ymaximum, zminimum, and zmaximumm which are used in the *boundingbox* definition.

```
SELECT projectxy(building.shape) FROM building
WHERE building.name = 'Chemistry'
```

3.1.3.3 Numerical Operations

We now move away from spatial operations and discuss numerical operations. Numerical operations produce a single numerical value as their result. They gather information about the statistics of a spatial object. The most commonly used numerical operations are *volume*, *area*, *surfacearea*, *perimeter*, *length*,operations that give the maximum coordinates and the minimum coordinates of an object in all three directions, *distance*, and *cardinality*.

Operations	Signature	Semantics
length	*line3D* $\rightarrow$ *real*	$\sum_{i=1}^{n} \int_{A_i} \sqrt{(\frac{dx}{dt_i})^2 + (\frac{dy}{dt_i})^2 + (\frac{dz}{dt_i})^2}\, dt_i$ where $x = f_i(t_i)$,$y = g_i(t_i)$, and $z = h_i(t_i)$ and $A_1, A_2, \ldots, A_n$ are the *curves* of the *blocks* of A.
perimeter	*surface* $\rightarrow$ *real*	$length(\partial A)$
area		$\sum_{i=1}^{n} \iint_{A_i} \sqrt{\gamma_{u_i}\gamma_{v_i} - (\frac{\partial x}{\partial u_i}\frac{\partial x}{\partial v_i} + \frac{\partial y}{\partial u_i}\frac{\partial y}{\partial v_i} + \frac{\partial z}{\partial u_i}\frac{\partial z}{\partial v_i})}\, du_i dv_i$ where $\gamma_w = ((\frac{\partial x}{\partial w})^2 + (\frac{\partial y}{\partial w})^2 + (\frac{\partial z}{\partial w})^2)$, $x = f_i(u_i, v_i)$,$y = g_i(u_i, v_i)$, and $z = h_i(u_i, v_i)$ and $A_1, A_2, \ldots, A_n$ are the *surface components* of A.
surfacearea	*volume* $\rightarrow$ *real*	$area(\partial A)$
volume		$\sum_{i=1}^{n} \iiint_{A_i} dx dy dz$ where $A_1, A_2, \ldots, A_n$ are the *simple volumes with cavities* of A.
xmaximum	$\alpha \rightarrow$ *real*	x_{max} where $(x_{max}, y, z) \in A \wedge \forall (x, y, z) \in A, x_{max} \geq x$
ymaximum		y_{max} where $(x, y_{max}, z) \in A \wedge \forall (x, y, z) \in A, y_{max} \geq y$
zmaximum		z_{max} where $(x, y, z_{max}) \in A \wedge \forall (x, y, z) \in A, z_{max} \geq z$
xminimum		x_{min} where $(x_{min}, y, z) \in A \wedge \forall (x, y, z) \in A, x_{min} \geq x$
yminimum		y_{min} where $(x, y_{min}, z) \in A \wedge \forall (x, y, z) \in A, y_{min} \geq y$
zminimum		z_{min} where $(x, y, z_{min}) \in A \wedge \forall (x, y, z) \in A, z_{min} \geq z$
distance	$\alpha_i \times \alpha_j$ $\rightarrow$ *real*	$\min\{d((x_a, y_a, z_a), (x_b, y_b, z_b)) \mid (x_a, y_a, z_a) \in A \wedge (x_b, y_b, z_b) \in B\}$
cardinality	$\alpha_0 \rightarrow int$	$\lvert A \rvert$
	$\alpha_1 \rightarrow int$	$\lvert blocks(A) \rvert$
	$\alpha_2 \rightarrow int$	$\lvert ss(A) \rvert$
	$\alpha_3 \rightarrow int$	$\lvert svc(A) \rvert$

Figure 3-15. Numerical operations.

Though the results that these operations produce are simple, the formulas required to perform many of them are complicated. This is because the continuous shapes and boundaries of 3D spatial objects means that only through use of calculus is it possible to concisely and rigorously specify the result of some of the operations. Also, we must sometimes deal with each of the components of a spatial object separately.

Calculus is involved in the calculation of *length*, *area*, and *volume*. These three operations compute the size of a spatial object. However, some of them are only

meaningful for certain spatial types. Specifically, they give the sizes of *line3D* objects, *surface* and *volumes* respectively. The definition of *length* uses an integral, of *area* uses a double integral, and of *volume* uses a triple integral. Furthermore, to get the respective results for each of these three operations, there is an integration over each of the components of an object separately and then a sum of the integrals of the components. The reader can see this in the semantics specifying the operations.

The operations *perimeter* and *surfacearea* produce information about the size of the boundary of a spatial object. The *perimeter* of a *surface*, however, is simply found by applying the *length* operation to its boundary. Similarly, the *surfacearea* of a *volume* is found by applying the *area* operation to the *volume*'s boundary.

Both the signatures and the semantics of the operations that calculate maximum and minimum coordinates are easy. The operations are *xmaximum*, *ymaximum*, *zmaximum*, *xminimum*, *yminimum*, and *zminimum*. These operations simply select the maximum and minimum coordinate values from the point set representing a spatial object. Maximum and minimum coordinate values may be calculated for an object of any 3D spatial data type. Moreover, the *distance* operation computes the minimum distance between two spatial object.

The *cardinality* operation determines the number of components of a spatial object. Thus, it determines the number of points in a *point3D* object, the number of *blocks* in a *line3D* object, the number of *simplesurface* (*ss*) in a *surface* or the number of *simplevolumeswithcavities* (*svc*) in a *volume*. It is the only operation that produces an integer as its result.

Figure 3-15 gives signatures and semantics for numerical operations. Recall again that d is a function representing the distance between two points.

Below are two examples (Q8 and Q9) illustrate queries for numerical operations.

Q8: Compute the area of the Physics building.

```
SELECT area (building.shape) FROM building
```

```
WHERE building.name = 'Physics'
```

Q9: Get total length of the sewer pipeline.

```
SELECT length(pipeline.shape) FROM pipeline
WHERE pipeline.type = 'sewer'
```

Operations play an important role in the *Spatial Algebra 3D* (*SPAL3D*). The operations can produce three different types of results. Each one of the operations yields another spatial object, a numerical value or a boolean value. The operations that yield another spatial object as their result, called *geometric operations*, breakdown further into geometric set operations, affine transformations, and other geometric operations. We will see that the design of the three-dimensional (3D) spatial data types promotes closure in the operations that result in another spatial object. The operations that result in a boolean value are, of course, spatial predicates. Two most important spatial predicates are the topological predicates like *inside* and the directional predicates like *north*. In recent years, great efforts have been spent on exploring possible topological relationships and directional relationships between two spatial objects. We discuss this in details in Section 3.2. In this section, we focus on the first two categories and only mention some spatial predicates.

3.2 Modeling Qualitative Relationships between Three-dimensional Objects

Besides all spatial operations we have discussed in Section 3.1.3, the category left is the spatial predicates. Predicates are important in queries because they describe whether two values satisfy a certain relationship, thus they are used frequently as filter conditions for selection queries and join queries. Spatial predicates are the ones that describe the spatial relationships between two spatial objects. Motivated by this, we explore the qualitative relationships between two 3D objects, which characterize the relative position of spatial objects like inside and below.

In this section, we explore two most important qualitative spatial relationships between two 3D objects, the topological relationships and the cardinal direction relationships.

3.2.1 3D Topological Relationships Modeling

The topological relationships between spatial objects are considered to be important for spatial databases. They lead to topological predicates, which can be embedded into spatial queries as join or selection conditions. Before rushing into the implementation of topological predicates, topological relationships between spatial objects must be first understood and clarified. This requires a detailed study of a vast number of possible spatial configurations at the abstract level, and as a result, methods that are able to classify and identify as many as possible different spatial configurations are needed. While a lot of research has already been carried out for topological relationships in the 2D space, the investigation in the 3D space are rather neglected. Developed modeling strategies are mostly extensions from the popular 9 intersection model (9IM) which has been originally designed for simple 2D spatial objects. We observe that a large number of topological relationships, especially the ones between two complex 3D objects are still not distinguished in these models.

We first review the 9IM based models and discuss the problems in Section 3.2.1.1. Then in Section 3.2.1.2, we proposal a new framework called *Neighborhood Configuration Model* (*NCM*) for modeling 3D topological relationships.

3.2.1.1 Problems with 9IM based 3D Topological Relationships

The central conceptual approach, upon which almost all publications in this field have been based, is the 9-intersection model [32]. This model checks the nine intersections of the *boundary*, *interior*, and *exterior* of a spatial object with the respective components of another spatial object for the topologically invariant criterion of non-emptiness. Extensions have been proposed to obtain more fine-grained topological predicates. However, the main focus has been on spatial objects in the 2D space; the

study of topological relationships between spatial objects in the 3D space has been rare. An available strategy is to apply 9IM based models and to investigate the total number of relationships that can occur in reality between spatial objects in 3D space. The review of the modeling strategies based on 9IM is presented in Section 2.4.1.

However, the third dimension introduces more complicated topological situations between 3D spatial objects. When directly applying the 9IM based models to 3D complex spatial objects like volumes, they suffer from a major problem, which we call the *high granularity* problem. That is, the 9IM considers the interior, exterior, and boundary point sets as the basic elements for empty and non-empty intersection tests, which ignores the fact that the interior, exterior, and the boundary of a spatial object are also complex spatial object parts, and may have multiple components. Thus the interaction between any pair of the basic elements from two spatial objects can be complex, and empty or non-empty may not be enough to describe such interactions. For example, the boundary of a volume object is a closed surface object, which may have multiple components. Thus the interaction between the boundaries of two volume objects is equivalent to the interaction between two complex surface objects, which can *touch* at a point, *meet* at a face, *cross* each other, or have *touch*, *meet*, and *cross* coexist on one or more components. Since 9IM based models do not have the capability to handle these interaction details for their basic elements, a large number of topological relationships between complex volumes are not distinguished.

3.2.1.2 The Neighborhood Configuration Model (NCM)

In this section, we are particular interested in complex volumes that may contain cavities or multiple components. A formal definition of complex volume objects can be found in Section 3.1.2.5, which models volumes as special infinite point sets in the three-dimensional Euclidean space. Our approach is also based on the point set theory and the point set topology. The basic idea is to evaluate the values of a set of boolean neighborhood configuration flags to determine the topological relationships between two

volumes. Each neighborhood configuration flag indicates the existence or non-existence of a characteristic neighborhood configuration of the points in a given scenario. The neighborhood configuration of a point describes the ownerships of the points that are "near" the reference point. If the existence of a neighborhood configuration is detected, then the corresponding neighborhood configuration flag is set to true. For example, for a scenario that involves two volumes A and B, if there exists a point p whose neighboring points all belong to both A and B, then the corresponding neighborhood configuration flag *exist_nei_in_overlap* (see Definition 27(1)) is set to true. Later, this neighborhood configuration flag contributes to the determination of the topological relationships between A and B.

Thus, two steps are involved in our approach. The first step is to explore all possible neighborhood configurations for a given scenario of two spatial volume objects, and the second step is to identified and encoded topological relationships with the neighborhood configuration flags.

- Step 1: Exploring neighborhood configuration information

In this section, we explore all neighborhood configurations. We begin with some needed topological and metric concepts. Let $\mathbb{R}$ be the set of real numbers, $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x > 0\}$, and $\mathbb{R}^3$ be the three-dimensional Euclidean space. Recall the Euclidean distance function $d : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ with $d(p, q) = d((x_1, y_1, z_1), (x_2, y_2, z_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$. Then for any point $p \in \mathbb{R}^3$ and $r \in \mathbb{R}^+$, the set $N_r(p) = \{q \in \mathbb{R}^3 \mid d(p, q) \leq r\}$ is called the closed ball with radius r and center p. Let ϵ denote a small positive value ($\epsilon \to 0$), we call the closed ball $N_\epsilon(p)$ the (closed) neighborhood of point p. We can think of the neighborhood around p as a tiny closed ball containing all points in $\mathbb{R}^3$ that are most "near" p.

With the necessary topological concepts introduced, we now explore the possible neighborhood configurations of points in a scenario that involves two volume objects. Let us assume two volumes A and B that are point sets in $\mathbb{R}^3$ ($A \subset \mathbb{R}^3, B \subset \mathbb{R}^3$). Then given

a point $p \in \mathbb{R}^3$, any point q in its neighborhood $N_\epsilon(p)$ ($q \in N_\epsilon(p)$) has one of the following four possible ownerships: (i) $q \in A \wedge q \notin B$, (ii) $q \notin A \wedge q \in B$, (iii) $q \in A \wedge q \in B$, and (iv) $q \notin A \wedge q \notin B$. As a result, we can describe the ownership of a neighborhood by combining the ownerships of all points in it. We call this ownership description of a neighborhood the neighborhood configuration. With the four possible ownerships of a single point, we can obtain a total of $C_4^1 + C_4^2 + C_4^3 + C_4^4 = 15$ possible ownership combinations for a neighborhood. In another word, 15 neighborhood configurations are possible for any point in $\mathbb{R}^3$ where A and B are embedded. As a result, we can define 15 corresponding neighborhood configuration flags for the scenario that involves A and B in Definition 27.

Definition 27. *Let $A \subset \mathbb{R}^3$ and $B \subset \mathbb{R}^3$, and let $p \in \mathbb{R}^3$ and $N_\epsilon(p)$ denote the neighborhood of p with a tiny radius, we first define four ownership flags for p:*

$$\alpha(A, B, p) \Leftrightarrow \exists x \in N_\epsilon(p) : x \in A \wedge x \in B$$

$$\beta(A, B, p) \Leftrightarrow \exists x \in N_\epsilon(p) : x \in A \wedge x \notin B$$

$$\gamma(A, B, p) \Leftrightarrow \exists x \in N_\epsilon(p) : x \notin A \wedge x \in B$$

$$\lambda(A, B, p) \Leftrightarrow \exists x \in N_\epsilon(p) : x \notin A \wedge x \notin B$$

Then we define the following 15 neighborhood configuration flags for a scenario involves two objects A and B:

(1) $exist\_nei\_in\_overlap(A, B)$

$$\Leftrightarrow \exists p \in \mathbb{R}^3 : \alpha(A, B, p) \wedge \neg\beta(A, B, p) \wedge \neg\gamma(A, B, p) \wedge \neg\lambda(A, B, p)$$

(2) $exist\_nei\_in\_op1(A, B)$

$$\Leftrightarrow \exists p \in \mathbb{R}^3 : \neg\alpha(A, B, p) \wedge \beta(A, B, p) \wedge \neg\gamma(A, B, p) \wedge \neg\lambda(A, B, p)$$

(3) $exist\_nei\_in\_op2(A, B)$

$$\Leftrightarrow \exists p \in \mathbb{R}^3 : \neg\alpha(A, B, p) \wedge \neg\beta(A, B, p) \wedge \gamma(A, B, p) \wedge \neg\lambda(A, B, p)$$

(4) $exist\_nei\_in\_ext(A, B)$

$$\Leftrightarrow \exists p \in \mathbb{R}^3 : \neg\alpha(A, B, p) \wedge \neg\beta(A, B, p) \wedge \neg\gamma(A, B, p) \wedge \lambda(A, B, p)$$

(5) $exist\_nei\_contain\_overlap\_op1(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \alpha(A, B, p) \wedge \beta(A, B, p) \wedge \neg\gamma(A, B, p) \wedge \neg\lambda(A, B, p)$

(6) $exist\_nei\_contain\_overlap\_op2(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \alpha(A, B, p) \wedge \neg\beta(A, B, p) \wedge \gamma(A, B, p) \wedge \neg\lambda(A, B, p)$

(7) $exist\_nei\_contain\_overlap\_ext(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \alpha(A, B, p) \wedge \neg\beta(A, B, p) \wedge \neg\gamma(A, B, p) \wedge \lambda(A, B, p)$

(8) $exist\_nei\_contain\_op1\_op2(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \neg\alpha(A, B, p) \wedge \beta(A, B, p) \wedge \gamma(A, B, p) \wedge \neg\lambda(A, B, p)$

(9) $exist\_nei\_contain\_op1\_ext(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \neg\alpha(A, B, p) \wedge \beta(A, B, p) \wedge \neg\gamma(A, B, p) \wedge \lambda(A, B, p)$

(10) $exist\_nei\_contain\_op2\_ext(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \neg\alpha(A, B, p) \wedge \neg\beta(A, B, p) \wedge \gamma(A, B, p) \wedge \lambda(A, B, p)$

(11) $exist\_nei\_contain\_op1\_op2\_ext(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \neg\alpha(A, B, p) \wedge \beta(A, B, p) \wedge \gamma(A, B, p) \wedge \lambda(A, B, p)$

(12) $exist\_nei\_contain\_op2\_overlap\_ext(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \alpha(A, B, p) \wedge \neg\beta(A, B, p) \wedge \gamma(A, B, p) \wedge \lambda(A, B, p)$

(13) $exist\_nei\_contain\_op1\_overlap\_ext(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \alpha(A, B, p) \wedge \beta(A, B, p) \wedge \neg\gamma(A, B, p) \wedge \lambda(A, B, p)$

(14) $exist\_nei\_contain\_op1\_op2\_overlap(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \alpha(A, B, p) \wedge \beta(A, B, p) \wedge \gamma(A, B, p) \wedge \neg\lambda(A, B, p)$

(15) $exist\_nei\_contain\_op1\_op2\_overlap\_ext(A, B)$

$\Leftrightarrow \exists p \in \mathbb{R}^3 : \alpha(A, B, p) \wedge \beta(A, B, p) \wedge \gamma(A, B, p) \wedge \lambda(A, B, p)$

The above 15 neighborhood configuration flags identify all possible interactions between two spatial volumes A and B in $\mathbb{R}^3$ at any single point. We demonstrate the validity of these neighborhood configuration flags by creating drawings for the corresponding neighborhood configurations in Figure 3-16. For example, if flag (8) yields true (Figure 3-16(8)), then it means that there exist a point whose neighborhood consists

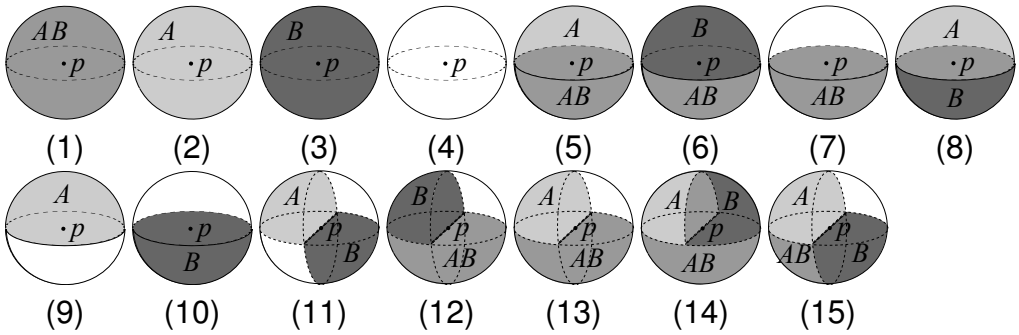


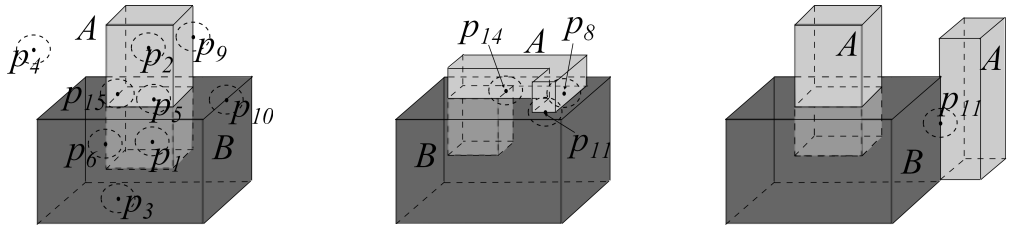
Figure 3-16. The drawing of the 15 neighborhood configurations for point p.

of points that are only from the first operand object A and points that are only from the second operand object B. The true value of this flag (*exist_nei_contain_op1_op2*) further indicates the existence of a meeting on face topological relationship between two volume objects A and B.

- Step 2: Topological relationship encoding with neighborhood configuration flags

We have introduced 15 neighborhood configuration flags for two complex volume objects embedded in the Euclidean space $\mathbb{R}^3$. These flags capture all topological situations for two volumes at all points in the Euclidean space $\mathbb{R}^3$. In other words, we have identified the topological relationships between two volumes at a very low granularity level, which involves a small neighborhood of a point. Thus, to determine the topological relationships between two volumes, we just need to collect the values of all 15 neighborhood configuration flags. First, let F [] denote an array that stores the value of all 15 neighborhood configuration flags that are defined in the previous section. Further we assume the ordering of the flags stored in F [] is the same as in Definition 27. Now, we are ready to define the topological relationship encoding for two volume objects.

Definition 28. *Let A, B be two volume objects in $\mathbb{R}^3$ ($A, B \subset \mathbb{R}^3$), and let FV denote the function that encodes the topological interaction between two volumes with respect to the ith neighborhood configuration flag's value ($1 \leq i \leq 15$). Then we have:*



(111111001100001) (111111011110011) (111111001110001)

(a) (b) (c)

Figure 3-17. Examples of topological relationship scenarios and their encodings.

$$FV(A, B, i) = \begin{cases} 0 \text{ if } \quad F[i] \text{ yields } false \text{ for } A \text{ and } B \\ 1 \text{ if } \quad F[i] \text{ yields } true \text{ for } A \text{ and } B \end{cases}$$

Thus, we can use a 15 bit binary array to encode the topological relationship between A and B. The definition for the topological relationship encoding TRE is given as:

$$TRE(A, B) = (FV(A, B, 0)\ FV(A, B, 1)\ \ldots\ FV(A, B, 14))$$

Definition 28 introduced a 15 bits binary representation for the topological relationships between two volume objects. Figure 3-17 presents three examples of the topological relationship encodings. The encoding in Figure 3-17a indicates that 9 topological relationship flags yield true for the scenario involving A and B, which are *exist_nei_in_overlap*, *exist_nei_in_op1*, *exist_nei_in_op2*, *exist_nei_in_ext*, *exist_nei_contain_overlap_op1*, *exist_nei_contain_overlap_op2*, *exist_nei_contain_op1_ext*, *exist_nei_contain_op2_ext*, and *exist_nei_contain_op1_op2_overlap_ext*. To demonstrate the validity of the encoding, we have marked the corresponding points p_i that validates the true value of flag $F[i]$. In Figure 3-17b, three additional flags, which are *exist_nei_contain_op1_op2*, *exist_nei_contain_op1_op2_ext*, and *exist_nei_contain_op1_op2_overlap*, become true due to the points p_8, p_{11}, and p_{14} respectively. Therefore, Figure 3-17 presents three different topological relationships that can be distinguished by our encoding, which are overlap encoded

by 111111001100001, overlap with meet on face encoded by 111111011110011, and overlap with meet on edge encoded by 111111001110001.

As a result, we obtain a total of $2^15 = 32768$ possible topological relationship encoding values, which implies a total of 32768 possible topological relationships between two complex objects. However, not all encoding values represent valid topological relationships. A simple example of an invalid encoding is the encoding 000000000000000. It is invalid because we can not associate a real word scenario with it. We call a topological relationship encoding valid for two objects if, and only if, the encoding can be derived from a real world scenario that involves two spatial objects. Thus the open questions are now (i) which and how many are the valid topological relationship encodings for two complex volumes (see Section 3.2.1.3), (ii) which and how many are the valid topological relationship encodings for two simple volumes (see Section 3.2.1.4). These two questions are important for three main reasons. First, we would like to get an indication of the expressiveness of our NCM model. At this point we know that 32768 encodings can be formed. But we do not know how many correspond to the possible real world scenarios of two complex/simple volumes that represent valid spatial configurations. Second, we would like to compare the number of valid encodings with the number of valid 9-intersection matrices from [31] and [106] for the complex/simple volumes. Third, since most current topological relationships reasoning models are based on the 9-intersection matrix model, we believe that the knowledge about the valid NCM topological encodings will lead to new insights of topological reasoning.

3.2.1.3 Determining Possible Topological Relationships Based on the NCM for Two Complex Volumes

In this section, we propose solutions to the first aforementioned question regarding which and how many are the valid topological relationship encodings for two complex volumes. A complex volume is a volume possibly with cavities or multiple volume

components, where isolated points, lines or surface parts are not allowed and the anomalies like missing points, missing lines, and missing surfaces parts are not allowed. A formal description about a complex volume object is given in [89]. In order to determine the valid topological relationship encodings, we employ a three-stage proof technique called proof-by-constraint-composition-drawing. In a first stage, we determine and prove the correctness of a collection of consistency constraints that can be used to filter out all impossible encodings that do not represent valid topological relationships between complex volumes. In a second stage, we identify a smaller set of topological encodings from the first stage that satisfy following two conditions (i) they can be used to compose all rest topological encodings from the first stage; (ii) they can not be composed by any other topological encodings. We call the topological encodings that satisfy these two conditions the unit topological encodings. Here, the composition operation refers to the logical or operation. In this stage, we also prove that any topological encoding that can be composed by unit encodings can be derived from a real world scenario that is composed by scenarios derived from those unit encodings. As a result, we only need to prove the validity of the unit encodings, so that all other encodings that can be composed from the unit encodings are also valid. In a third stage, we verify the correctness of the unit topological encodings identified from the second stage by drawing prototypical spatial configurations.

- The first stage: filtering out the invalid topological encodings

We begin with the first stage, that is, the specification of the constraints for filtering invalid encodings. We observe that the 15 neighborhood configurations are not completely independent from each other. There exist correlations between some configurations. For example, the existence of the neighborhood configuration (5) in Figure 3-16(5) implies the existence of both the neighborhood configuration (1) (Figure 3-16(1)) and the neighborhood configuration (2) (Figure 3-16(2)). Thus, we explore these correlations between the 15 topological relationships and use them to

identify the constraints. Let E be a 15 bit array that stores the topological encoding. Thus we have for any bit E_i of E, with $1 \leq i \leq 15$, $E_i \in 0, 1$. Let NC_i with $1 \leq i \leq 15$ refers to the neighborhood configuration (i) in Figure 3-16(i). If E is a valid 15 bit topological encoding for two complex volume objects A and B, it must satisfy the constraints specified in Lemma 1 to Lemma 5.

Lemma 1. *There always exist a neighborhood configuration that only contains points belonging to neither A nor B, i.e.,*

$$E_4 \equiv 1$$

Proof. The proof for this lemma is trivial. Since volume objects A and B in the 3D space are bounded, $\mathbb{R}^3 - A - B \neq \emptyset$. □

Lemma 2. *(i) If, and only if NC_1 exists, at least one of the following seven neighborhood configurations, $NC_5, NC_6, NC_7, NC_{12}, NC_{13}, NC_{14}, NC_{15}$ must exist; (ii) if, and only if NC_2 exists, at least one of the following seven neighborhood configurations, $NC_5, NC_8, NC_9, NC_{11}, NC_{13}, NC_{14}, NC_{15}$ must exist; (iii) if, and only if NC_3 exists, at least one of the following seven neighborhood configurations, $NC_6, NC_8, NC_{10}, NC_{11}, NC_{12}, NC_{14}, NC_{15}$ must exist; (iv) if, and only if NC_4 exists, at least one of the following seven neighborhood configurations, NC_7, NC_9, NC_{10},$NC_{11}, NC_{12}, NC_{13}, NC_{15}$ must exist, i.e.,*

$$(i)\ E_1 = 1 \Leftrightarrow \exists i \in \{5, 6, 7, 12, 13, 14, 15\} : E_i = 1$$

$$(ii)\ E_2 = 1 \Leftrightarrow \exists i \in \{5, 8, 9, 11, 13, 14, 15\} : E_i = 1$$

$$(iii)\ E_3 = 1 \Leftrightarrow \exists i \in \{6, 8, 10, 11, 12, 14, 15\} : E_i = 1$$

$$(iv)\ E_4 = 1 \Leftrightarrow \exists i \in \{7, 9, 10, 11, 12, 13, 15\} : E_i = 1$$

Proof. First, we prove Lemma 2 (i), that is, if NC_1 exists at least one of the seven neighborhood configurations, $NC_5, NC_6, NC_7, NC_{12}, NC_{13}, NC_{14}, NC_{15}$ must exist. For this purpose, we first introduce some needed concepts in point-set topology. Let $X \subset \mathbb{R}^3$ and $p \in \mathbb{R}^3$. p is an interior point of X if there exists a neighborhood $N_r(p)$ such that $N_r(p) \subseteq X$. p is an exterior point of X if there exists a neighborhood $N_r(p)$ such that $N_r(p) \cap X = \emptyset$. p is a boundary point of X if p is neither an interior nor exterior point of

X. The set of all interior / exterior / boundary points of X is called the *interior* / *exterior* / *boundary* of X and is denoted by X° / X^- / ∂X. For a volume object V in the 3D space, we have $V^\circ \neq \emptyset \wedge V^- \neq \emptyset \wedge \partial V \neq \emptyset$. So if we assume that for two volume objects A and B, NC_1 exists, but none of the seven configurations, NC_5, NC_6, NC_7, NC_{12}, NC_{13}, NC_{14}, NC_{15} exist, then, for any point $p \in \mathbb{R}^3$, its neighborhood $N_r(p)$ is either $N_r(p) \subseteq A$ (p is an *interior* point of A) or $N_r(p) \cap A = \emptyset$ (p is an *exterior* point of A). As a result, there does not exist a boundary point of A. This is a contradiction to the requirement that any volume object must have interior points, boundary points and exterior points. Hence, the assumption is invalid. Now we prove that if at least one of the seven neighborhood configurations, NC_5, NC_6, NC_7, NC_{12}, NC_{13}, NC_{14}, NC_{15} exists, then NC_1 must exist.
If NC_5 exists, then there exists a point p with a neighborhood $N_r(p)$ such that $N_r(p)$ contains points belonging only to A and points belonging to both A and B (Definition 27). So there exists a set $X \subset N_r(p)$ so that X contains only points from A. Then any point $q \in X$ has a neighborhood that belongs only to A. Hence, NC_1 exists. Similarly, we can prove the existence of NC_1 for the other six neighborhood configurations. Similarly, the rest of Lemma 2, Lemma 2 (ii) to Lemma 2 (iv) can be proved in a same way. □

Lemma 3. *If the neighborhood configuration (1) does not exist, then there must exist neighborhood configuration (2) and (3), i.e.,*

$$E_1 = 0 \Rightarrow E_2 = 1 \wedge E_3 = 1$$

Proof. According to Lemma 2 (i), if NC_1 does not exists ($E_1 = 0$), then neighborhood configurations NC_5, NC_6, NC_7, NC_{12}, NC_{13}, NC_{14}, and NC_{15} do not exist. Further, if we assume that NC_2 does not exist ($E_2 = 0$), then according to Lemma 2 (ii), the neighborhood configurations NC_5, NC_8, NC_9, NC_{11}, NC_{13}, NC_{14}, and NC_{15} do not exist. As a result, only the neighborhood configurations NC_3, NC_4, and NC_{10} possibly exist. However, these neighborhood configurations do not contain any points that are from volume object A, thus violates the requirement that both volume object A and volume

object B must exist. Hence, the assumption is invalid. Similarly, we can prove if NC_1 does not exists ($E_1 = 0$) then NC_3 must exist ($E_3 = 1$). □

Lemma 4. *(i) If NC_{11} exists, at least two of the following neighborhood configurations, NC_8, NC_9, NC_{10} must exist; (ii) if NC_{12} exists, at least two of the following neighborhood configurations, NC_6, NC_7, NC_{10} must exist; (iii) if NC_{13} exists, at least two of the following neighborhood configurations, NC_5, NC_7, NC_9 must exists; (iv) if NC_{14} exists, at least one of the following neighborhood configurations, NC_5, NC_6, NC_8 must exists, i.e.,*

$$(i)\ E_{11} = 1 \Rightarrow (E_8 = 1 \vee E_9 = 1) \wedge (E_8 = 1 \vee E_{10} = 1) \wedge (E_9 = 1 \vee E_{10} = 1)$$

$$(ii)\ E_{12} = 1 \Rightarrow (E_6 = 1 \vee E_7 = 1) \wedge (E_6 = 1 \vee E_{10} = 1) \wedge (E_7 = 1 \vee E_{10} = 1)$$

$$(iii)\ E_{13} = 1 \Rightarrow (E_5 = 1 \vee E_7 = 1) \wedge (E_5 = 1 \vee E_9 = 1) \wedge (E_7 = 1 \vee E_9 = 1)$$

$$(iv)\ E_{14} = 1 \Rightarrow (E_5 = 1 \vee E_6 = 1) \wedge (E_5 = 1 \vee E_8 = 1) \wedge (E_6 = 1 \vee E_8 = 1)$$

Proof. First, we prove Lemma 4 (i), that is, if NC_{11} exists, at least two of the following neighborhood configurations, NC_8, NC_9, NC_{10} must exist. According to the definition of the neighborhood configuration (11) in Definition 27, if NC_{11} exists ($E_{11} = 1$), then there exists a open ball B in the 3D space that contains three point sets, the set S_a that contains points that are only from volume object A ($S_a \subset A$), the set S_b that contains points that are only from B ($S_b \subset B$), and the set S_x that contains points that are neither from A nor from B. So we have $B = S_a \cup S_b \cup S_x$ and $S_a^\circ \cap S_b^\circ \cap S_x^\circ = \emptyset$. Since B is a closed ball in the 3D space, its interior is connected. Thus ∂S_a must meet with ∂S_b or ∂S_x, and ∂S_b must meet with ∂S_a or ∂S_x, and ∂S_x must meet with ∂S_a or ∂S_b. This results in the existence of at least two of the neighborhood configurations NC_8, NC_9, and NC_{10}. The rest of Lemma 4, Lemma 4(ii) to Lemma 4(iv), can be proved in a similar manner. □

Lemma 5. *If neighborhood configurations NC_1, NC_2, NC_3, and NC_4 exist, then at least three of the six neighborhood configurations, $NC_5, NC_6, NC_7, NC_8, NC_9, NC_{10}$, at least one of the three neighborhood configurations, NC_5, NC_6, NC_7, at least one of the*

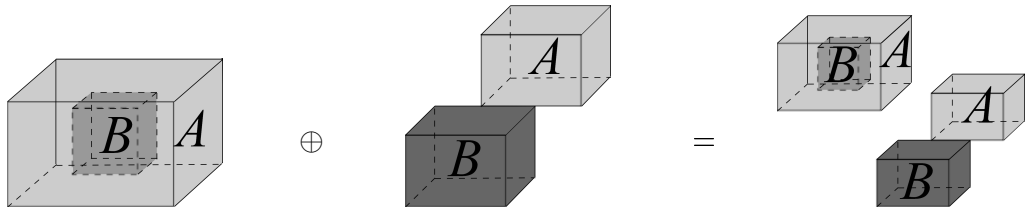
three configurations, NC_5, NC_8, NC_9*, at least one of the three neighborhood configurations,* NC_6, NC_8, NC_{10}*, and at least one of the three neighborhood configurations,* NC_7, NC_9, NC_{10}*, must exist i.e.,*

$$E_1 = 1 \wedge E_2 = 1 \wedge E_3 = 1 \wedge E_4 = 1 \Rightarrow$$
$$(E_5 = 1 \vee E_6 = 1 \vee E_7 = 1) \wedge (E_5 = 1 \vee E_8 = 1 \vee E_9 = 1) \wedge$$
$$(E_6 = 1 \vee E_8 = 1 \vee E_{10} = 1) \wedge (E_7 = 1 \vee E_9 = 1 \vee E_{10} = 1) \wedge$$
$$\exists i, j, k \in \{5, 6, 7, 8, 9, 10\}, i \neq j \neq k : E_i = 1 \wedge E_j = 1 \wedge E_k = 1$$

Proof. Let $S_a, S_b, S_{ab}, S_x \subset \mathbb{R}^3$ with $S_a = A - B$, $S_b = B - A$, $S_{ab} = A \cap B$, and $S_x = \mathbb{R}^3 - A - B$. Then we have $S_a \cup S_b \cup S_{ab} \cup S_x = \mathbb{R}^3$. If neighborhood configurations NC_1, NC_2, NC_3, and NC_4 exist, then we have $S_a \neq \emptyset$, $S_b \neq \emptyset$, $S_{ab} \neq \emptyset$, and $S_x \neq \emptyset$. So these four sets must meet with each other to form the 3D space. So for S_{ab} it must meet with at least one of the other set, S_a or S_b or S_x. As a result, at least one of the neighborhood configurations, NC_5, NC_6, and NC_7 must exist $((E_5 = 1 \vee E_6 = 1 \vee E_7 = 1))$. Similarly, we can prove $E_5 = 1 \vee E_8 = 1 \vee E_9 = 1$, $E_6 = 1 \vee E_8 = 1 \vee E_{10} = 1$, and $E_7 = 1 \vee E_9 = 1 \vee E_{10s} = 1$. Further, we can not have every one of the four sets only meet with one other set, otherwise the four sets cannot form the entire 3D space. As a result, at least three of the neighborhood configurations $NC_5, NC_6, NC_7, NC_8, NC_9$, and NC_{10}, must exist. □

So far in this first stage, we have identified five constraints that every valid topological relationship encodings for two complex volume objects must satisfy. We have also proved the correctness of the constraints so that we ensure that the encodings that do not satisfy these constraints are invalid ones. We have developed a simple program that checks the validity of all 32768 (2^15) 15 bit encodings with respect to the constraints, from Lemma 1 to Lemma 5. As a result, we get a total of 697 encodings that satisfy all above five constraints.

- The second stage: identifying the unit topological encodings



110110001000000 | 011100001110000 = 111110001110000

Figure 3-18. The composition of two encodings implies the composition of two configurations.

Our remaining task is to show the completeness of the constraints (that is, no further constraint is missing) and the correctness of the remaining encodings (that is, the remaining encodings are all valid). A simple strategy is to show for each remaining encoding that a spatial configuration of two complex volume objects A and B exists which represents a real world scenario and can be mapped to the encoding. However, the creation of spatial configuration drawings for all 697 encodings is a tedious effort and is unnecessary. Thus it is the task of this stage to reduce the number of encodings that need to be drawn. For this purpose, we explore the relevance among the encodings. We observe that some encodings can be composed by some other encodings by applying a logical *or* operation (| operation). For example, the encoding 111110001110000 can be composed by applying the logical or operation $|$ on the following two encodings, 110110001000000 and 011100001110000, so we have $111110001110000 = 110110001000000|011100001110000$. Further, we prove that if an encoding E can be composed by encodings $E_1, \ldots, E_m$, i.e., $E = E_1|\ldots|E_m$ then the spatial configuration that can prove the correctness of the encoding E can also be drawn by composing the spatial configurations of the encodings $E_1, \ldots, E_m$. We give the definition for such a spatial composition operation in Definition 29

Definition 29. *Let* $S_1 = \langle A_1, B_1 \rangle$ *and* $S_2 = \langle A_2, B_2 \rangle$ *with* $A_1, B_1, A_2, B_2 \subset \mathbb{R}^3$ *and* $A_1 \cap A_2 = A_1 \cap B_2 = B_1 \cap A_2 = B_1 \cap B_2 = \emptyset$ *denote two separate spatial configurations*

where each contains a pair of spatial volume objects. Let $S' = \langle A', B' \rangle$ *be another spatial configuration. Then the spatial composition operation* $\oplus$ *is defined as follows:*

$$S' = S_1 \oplus S_2 \quad \stackrel{\text{def}}{\Leftrightarrow} \quad (A' = A_1 \cup A_2) \wedge (B' = B_1 \cup B_2)$$

Lemma 6 shows that the logical or of the configuration encodings implies the spatial composition of corresponding spatial configurations.

Lemma 6. *Let* S_1 *and* S_2 *denote two separate spatial configurations. Let the function* tpE *denote the function that maps a spatial configuration to a topological relationship encoding. Then we have,*

$$tpE(S_1 \oplus S_2) = tpE(S_1) | tpE(S_2)$$

Proof. An important property of the composition operation (see Definition 29) is that no new topological relationships are introduced while composing two separated spatial configurations. As a result, the existence of the spatial configurations in the composed spatial configuration are all inherited from the two operand spatial configurations. In other words, an neighborhood configuration does not exist in the composed spatial configuration if the same neighborhood configuration does not exist in either operand spatial configuration. Hence, the topological encoding for the composed spatial configuration is the logical or of the encodings for the two operand spatial configurations. □

For the previous example, we can make the spatial configuration drawings S_1 and S_2 for the encodings 110110001000000 and 011100001110000 respectively to prove the correctness of these two encodings. If we compose S_1 and S_2, then according to Lemma 6, the encoding for the result spatial configuration is the result of $110110001000000|011100001110000$, which is the encoding 111110001110000. As a result, by composing S_1 and S_2, we can present a spatial configuration drawing for 111110001110000, which proves the correctness of the encoding 111110001110000. Figure 3-18 shows how the composition of two encodings implies the composition of two spatial configurations.

Now, given the set of 697 encodings from the first stage, we do not have to draw the spatial configurations for every encoding to prove their correctness, instead, we only need to find out a collection of topological relationship encodings in these 697 encodings that can be used to compose the rest of the encodings. Let the set TP denote the set containing 697 topological relationship encodings. Then we call the set $TPU = \{u_1, u_2, ..., u_n\}$ with $u_i \in TP$ $(1 \leq i \leq n)$ the set of unit topological relationship encodings if it satisfies the following two conditions: (i) for any encoding e in TP but not TPU ($e \in TP - TPU$), there exists a set $C \subseteq TPU, C = \{c_1, ..., c_m\}$ with $1 < m \leq n$ such that $e = c_1|c_2|...|c_m$; (ii) for any encoding u in TPU ($u \in TPU$), there does not exist a set a set $C \subseteq TP, C = \{c_1, ..., c_m\}$ with $1 < m \leq n$ such that $u = c_1|c_2|...|c_m$. The first condition implies that every non-unit encoding can be composed by some unit topological relationship encodings. The second condition implies that every unit encoding can not be composed by any other encodings.

We now present an algorithm for finding such a set of unit topological relationship encodings. The algorithm takes an array EL as input which keeps 697 records and whose index range represents the encoding id 1 to 697. Each record stores an integer whose binary value is a topological encoding. Our algorithm exams every topological relationship encoding against other 696 encodings. For each encoding $EL[i]$, we check if it can be composed by the rest of the encodings. If after we traversed the rest encodings, we still can not compose $EL[i]$, then $EL[i]$ is identified as a unit encoding, and it is pushed to the result array UL. Eventually, we can get all unit topological encodings and their corresponding encoding ids in the original encoding list EL. In total, we obtain 50 unit topological encodings.

- The third stage: verifying the correctness of the unit topological encodings

In the second stage, we have identified a total of 50 unit topological relationship encodings. According to Lemma 6, we can always construct a spatial configuration for any non-unit encoding E by composing the spatial configurations of the unit encodings

```
     algorithm DetermineNCMUnitEncodings(EL)
     input An array EL of 697 records. Each index between 1 and 697 is an encoding id.
           Each record stores an integer whose binary value equals a topological
           relationship encoding.
     output An array UL of the encoding id and the encoding integer value pair, who
           are identified as unit encodings.
(1)    i ← 1; j ← 1; n ← 697; ucnt ← 0; UL ← empty list
(2)    for each i in 1 ... n do
(3)      encoding ← EL[i]; isComposable ← false; currentCompositeVal ← 0;
(4)      for each j in 1 ... n do
(5)        c ← EL[j];
(6)        if i != j then
(7)          if encoding|c = encoding then  //a potential composable encoding
(8)            currentCompositeVal = currentCompositeVal|c;
(9)            if currentCompositeVal = encoding then
(10)             isComposable ← true;
(11)             break;
(12)           endif;
(13)         endif;
(14)       endif;
(15)     endfor;
(16)     if isComposable = false then  // found a unit encoding
(17)       ucnt++;
(18)       UL[ucnt] ← (i, encoding);
(19)     endif;
(20)   endfor;
(21)   return UL;
     end DetermineNCMUnitEncodings.
```

Figure 3-19. The algorithm to determine collection of unit topological relationship encodings.

$u_1, u_2, \ldots, u_m$ ($m \leq 50$) such that $E = u_1|u_2|\ldots|u_m$. As a result, in order to perform the validation of all 697 topological relationship encodings, we only need to draw prototypical spatial configurations in $\mathbb{R}^3$ for the 50 unit encodings. If we can show that the unit encodings are all valid, then we know that all encodings that can be composed from this set of unit encodings are also valid. It is a huge reduction in terms of the number of spatial configuration drawings for the validation process. We present all 50 spatial configuration drawings together with their corresponding unit topological relationship encodings and the corresponding 9IM matrices in Figure A-1 to Figure A-3.

Finally, we can conclude that our NCM model is able to distinguish a total of 697 different topological relationships between two complex volume objects in the 3D space.

3.2.1.4 Determining Possible Topological Relationships Based on the NCM for Two Simple Volumes

We have identified all valid topological relationships between two complex volumes that may contain cavities or multiple components. In this section, we would like to find out how many of these topological relationships are still valid between two simple volumes. Due to their simple structure, simple volume objects have always attracted particular interest among researchers in spatial information science. As we have seen in Section 2.4.1, most available topological relationship models are based on simple spatial objects. Although our model is more general, we now consider the case of simple volumes as a special case of complex regions in order to be able to compare our approach to others.

A simple volume object is defined as a bounded, regular closed set homeomorphic (that is, topologically equivalent) to a closed ball in $\mathbb{R}^3$. A simple volume object is also often called as a 3D-manifold. This, in particular, means that it has a connected interior, a connected boundary, and a connected exterior, Hence, it is not allowed to consist of several components, it must not contain non-manifold points, and it must not have cavities. Due to these unique properties of a simple volume object, besides the constraints in Lemma 1 to Lemma 5, additional constraints are needed to identify the encodings among the 697 topological relationship encodings for two complex volume objects that are also valid for two simple volume objects.

One most important characteristic property of a simple volume object is the path connectedness of its interior, exterior, and boundary. We have given the description about the interior, the exterior and the boundary of a given point set in $\mathbb{R}^3$ in the proof for Lemma 2. We say a point set S ($S \subset \mathbb{R}^3$) is path connected if, and only if for any two points $x, y \in S$, there exists a path L joining the two points x and y, where the path L is

a continuous function $f : \mathbb{R} \to \mathbb{R}^3$ from the unit interval $[0, 1]$ to the set S with $f(0) = x$ and $f(1) = y$ (Let $\mathbb{R}$ be the set of real numbers). As a result, any point on the path L belongs to the set S. Hence, given a simple volume object A, its interior A°, exterior A^- and boundary ∂A are all path connected sets. Based on this property, we give the additional constraints for two simple volumes A and B in Lemma 7 to Lemma 11.

Lemma 7. *(i) If neighborhood configuration NC_2 exists, then there must exist neighborhood configuration NC_9; (ii) if neighborhood configuration NC_3 exists, then there must exist neighborhood configuration NC_{10}, i.e.,*

(i) $E_2 = 1 \Rightarrow E_9 = 1$

(ii) $E_3 = 1 \Rightarrow E_{10} = 1$

Proof. First we prove (i), that is, if NC_2 exists ($E_2 = 1$), NC_9 exists ($E_9 = 1$). Since both A and B exist, the interior, the exterior, and the boundary of both volume objects exist. Given any point $x \in B^-$, the neighborhood configuration on x can only be one of the following three configurations: NC_2, NC_4, and NC_9. If NC_2 exists, then there exists a point q such that $q \in A^\circ \wedge q \in B^-$. According to Lemma 1, NC_4 always exists, which means that there alway exist a point p such that $p \in A^- \wedge p \in B^-$. So p, q both exist and they all belong to the exterior of B. Since the exterior of B (B^-) is a path connected set, thus there must exist a path L connecting p and q such that every point on L is in the exterior of B. Further, since p is in the interior of A and q is in the exterior of A, the path L that connects q and p must cross the boundary of A. In other words, there must exist a point m such that $m \in L$ and m is on the boundary of A. Since m is on the path L, m is in the exterior of B. As a result, the neighborhood configuration on m can only be one of the following three: NC_2, NC_4, and NC_9. NC_2 and NC_4 can be excluded for being the neighborhood configuration on m because m is neither in the interior of A (NC_2) nor the exterior of A (NC_4). Hence, the neighborhood configuration on m can only be NC_9, which proves the existence of neighborhood configuration (9) (NC_9). In a similar way, we can prove (ii). □

Lemma 8. *(i) If neighborhood configuration NC_{11} exists, then there must exist both neighborhood configuration NC_9 and NC_{10}; (ii) if neighborhood configuration NC_{12} exists, then there must exist both neighborhood configuration NC_6 and NC_{10}; (iii) if neighborhood configuration NC_{13} exists, then there must exist both neighborhood configuration NC_5 and NC_9; (iv) if neighborhood configuration NC_{14} exists, then there must exist both neighborhood configuration NC_5 and NC_6; (v) if neighborhood configuration NC_{15} exists, then there must exist neighborhood configurations NC_5, NC_6, NC_9, and NC_{10} i.e.,*

(i) $E_{11} = 1 \Rightarrow E_9 = 1 \wedge E_{10} = 1$

(ii) $E_{12} = 1 \Rightarrow E_6 = 1 \wedge E_{10} = 1$

(iii) $E_{13} = 1 \Rightarrow E_5 = 1 \wedge E_9 = 1$

(iv) $E_{14} = 1 \Rightarrow E_5 = 1 \wedge E_6 = 1$

(v) $E_{15} = 1 \Rightarrow E_5 = 1 \wedge E_6 = 1 \wedge E_9 = 1 \wedge E_{10} = 1$

Proof. The first statement is easy to prove. According to Lemma 2 (ii) and Lemma 2 (iii), if NC_{11} exists, then NC_2 and NC_3 must exist, which further indicates the existence of NC_9 and NC_{10} (see Lemma 7). We then prove (ii), that is if neighborhood configuration NC_{12} exists ($E_{12} = 1$), then there must exist both neighborhood configuration NC_6 and NC_{10} ($E_6 = 1 \wedge E_{10} = 1$). If NC_{12} exists, to prove the existence of NC_{10} is simple. According to Lemma 2 (iii), if NC_{12} exists, then NC_3 exists, which implies the existence of NC_{10} (see Lemma 7 (ii)). Now, we prove that the existence of NC_{12} implies the existence of NC_6. If NC_{12} exists, then according to the definition of the neighborhood configuration (12), there exists a set S that contains points only from B, points from both A and B, and points from neither A nor B. Let the sets $S_b, S_{ab}, S_x \subset S$ denote the set that contains points only from B, the set that contains points from both A and B, and the set that contains points from neither A nor B respectively. Then we have $S_b \cup S_{ab} \cup S_x = S$, $S_b^\circ, S_{ab}^\circ \subset B^\circ$, $S_{ab}^\circ \subset A^\circ$, and $S_b^\circ \subset A^-$. Since the interior of B is path connected, then given two points p, q such that $p \in S_b^\circ$ and $q \in S_{ab}^\circ$, there exists a path L that connects p and q, and L still lies in the interior of B ($L \subset B^\circ$). However, since p is in the exterior

of A and q is in the interior of A, L that connects p, q must cross the boundary of A. Let m denote the intersection point between the path L and ∂A, then the neighborhood of m is still in the interior of B ($N_\epsilon(m) \subseteq B^\circ$) because m is an interior point of B, and the neighborhood of m also contains points from the interior of A ($N_\epsilon(m) \cap A^\circ \neq \emptyset$) because m is on the boundary of A. Therefore, the neighborhood configuration on m is NC_6. Hence, we have proved the existence of NC_6. Similarly, we can prove the correctness of (iii), (iv), and (v). □

The neighborhood configuration NC_{11} describes a neighborhood $N_\epsilon(p)$ of point p that consists of three points sets, one from the volume object A denoted as S_a, one from the volume object B denoted as S_b, and the other from the exterior of both A and B denoted as S_x. We have $N_\epsilon(p) = S_a \cup S_b \cup S_x$ and $S_a^\circ \cap S_b^\circ \cap S_x^\circ = \emptyset$. Further, we observe that different arrangement of the three point sets in NC_{11} are possible according to the different interactions among them. For example, Figure 3-16(11) shows one possible arrangement of the three sets, where S_x is a disconnected set and it meets with both S_a and S_b on boundary face, while S_a and S_b do not meet on face. More arrangements of the three sets are possible. For example S_a can meet only with S_b on face but not with S_x (Figure 3-20C). However, according to Lemma 8 (i), at lease two out of the three neighborhood configurations, NC_8, NC_9, and NC_{10}, must exist in $N_\epsilon(p)$, that is, every set must meet with at least one other set. Thus, a total of four possible arrangements among the three sets are possible, we call these arrangements the neighborhood configuration variants for NC_{11}. Figure 3-20 demonstrates the four arrangements. The neighborhood configuration variants further distinguishes the different interactions among the three sets for NC_{11}, thus are more detailed neighborhood configurations. However, not all arrangements are valid arrangements for simple volumes. According to Lemma 8 (i), neighborhood configuration NC_9 and NC_{10} must exist in $N_\epsilon(p)$, which means that S_a and S_b must both meet S_x on face. As a result, only the two variants in Figure 3-20A and Figure 3-20B are valid arrangements of the three sets for two

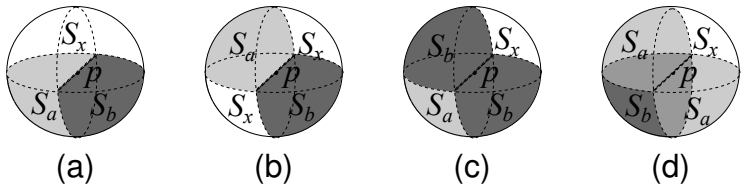


Figure 3-20. Examples of the four possible arrangements of the three sets S_a, S_b, and S_x in the neighborhood configuration NC_{11}.

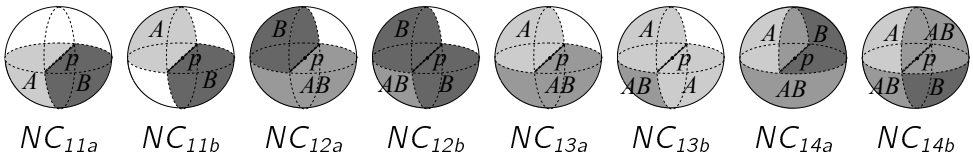


Figure 3-21. Drawings of the variants of the neighborhood configurations NC_{11}, NC_{12}, NC_{13}, and NC_{14}.

simple volumes A and B. The arrangements in Figure 3-20C and Figure 3-20D are valid arrangements for two complex volumes but invalid for two simple volumes. Similarly, we can find the variants for neighborhood configurations NC_{12}, NC_{13}, and NC_{14}. Definition 30 gives a formal definition for these neighborhood configuration variants.

Definition 30. *Let $A, B \subset \mathbb{R}^3$, $p \in \mathbb{R}^3$ and $N_\epsilon(p)$ denote the neighborhood of p with a tiny radius, and let $getnc$ denote the function that determines the neighborhood configuration on a given point p. Then we define for each neighborhood configuration NC_i with $i \in \{11, 12, 13, 14\}$ two neighborhood configuration variants NC_{ia} and NC_{ib}:*

$$NC_{11a} \Leftrightarrow \exists p \in \mathbb{R}^3 : getnc(p) = NC_{11} \wedge (\exists q \in N_\epsilon(p) : getnc(q) = NC_8)$$

$$NC_{11b} \Leftrightarrow \exists p \in \mathbb{R}^3 : getnc(p) = NC_{11} \wedge (\neg\exists q \in N_\epsilon(p) : getnc(q) = NC_8)$$

$$NC_{12a} \Leftrightarrow \exists p \in \mathbb{R}^3 : getnc(p) = NC_{12} \wedge (\exists q \in N_\epsilon(p) : getnc(q) = NC_7)$$

$$NC_{12b} \Leftrightarrow \exists p \in \mathbb{R}^3 : getnc(p) = NC_{12} \wedge (\neg\exists q \in N_\epsilon(p) : getnc(q) = NC_7)$$

$$NC_{13a} \Leftrightarrow \exists p \in \mathbb{R}^3 : getnc(p) = NC_{13} \wedge (\exists q \in N_\epsilon(p) : getnc(q) = NC_7)$$

$$NC_{13b} \Leftrightarrow \exists p \in \mathbb{R}^3 : getnc(p) = NC_{13} \wedge (\neg\exists q \in N_\epsilon(p) : getnc(q) = NC_7)$$

$$NC_{14a} \Leftrightarrow \exists p \in \mathbb{R}^3 : getnc(p) = NC_{14} \wedge (\exists q \in N_\epsilon(p) : getnc(q) = NC_8)$$

$$NC_{14b} \Leftrightarrow \exists p \in \mathbb{R}^3 : getnc(p) = NC_{14} \wedge (\neg\exists q \in N_\epsilon(p) : getnc(q) = NC_8)$$

For two simple volumes, we replace the original neighborhood configurations NC_{11}, NC_{12}, NC_{13} and NC_{14} with their variants which describe the interactions between the two volumes in a neighborhood in more details. Figure 3-21 shows the two variants for each of the four neighborhood configurations. Now, with the new configurations defined, we introduce an additional constraint in Lemma 9.

Lemma 9. *(i) If NC_i with $11 \leq i \leq 14$ exists, then one of its neighborhood configuration variants, NC_{ia} and NC_{ib}, must exist; (ii) if NC_8 does not exist, then neither NC_{11a} nor NC_{14a} exist; (iii) if NC_7 does not exist, then neither NC_{12a} nor NC_{13a} exist; i.e.,*

(i) $E_i = 1 \Leftrightarrow NC_{ia} \vee NC_{ib} \qquad (i \in \{11, 12, 13, 14\})$

(ii) $E_8 = 0 \Rightarrow \neg NC_{11a} \wedge \neg NC_{14a}$

(iii) $E_7 = 0 \Rightarrow \neg NC_{12a} \wedge \neg NC_{13a}$

Proof. The proof for i is trivial. We take NC_{11} for example. According to Definition 30, $NC_{ia} \vee NC_{ib}$ is equivalent to $(\exists p \in \mathbb{R}^3 : getnc(p) = NC_{11} \wedge (\exists q \in N_\epsilon(p) : getnc(q) = NC_8))$ $\vee (\exists p \in \mathbb{R}^3 : getnc(p) = NC_{11} \wedge (\neg\exists q \in N_\epsilon(p) : getnc(q) = NC_8))$, which is further equivalent to $\exists p \in \mathbb{R}^3 : getnc(p) = NC_{11}$. Thus (i) is proved. Similarly, we can prove the correctness of (i) for NC_{12}, NC_{13} and NC_{14}. Now, we prove (ii). According to Definition 30 for NC_{11a} and NC_{14a}, one condition for them to exist is the existence of the neighborhood configuration NC_8. Thus if NC_8 does not exist ($E_8 = 0$), neither NC_{11a} nor NC_{14a} exists. Similarly, we can prove that if NC_7 does not exist ($E_7 = 0$), neither NC_{12a} nor NC_{13a} exists. □

So far, we have introduced neighborhood configuration variants and the constraints that deals with these variants. Now, we introduce the last constraint for validating the topological encodings for two simple volumes. For this purpose, it is necessary to introduce an important concept called the boundary neighborhood configuration transition graph. We first define the concept of neighborhood configuration transition between two points in the 3D space.

Definition 31. *Let $p, q \in \mathbb{R}^3$ be two points in the 3D space. We call a link L that connects point p and q a path from p to q if and only if the path L is a continuous function $f : \mathbb{R} \to \mathbb{R}^3$ from the unit interval $[0, 1]$ to the 3D space with $f(0) = p$ and $f(1) = q$ (Let $\mathbb{R}$ be the set of real numbers). Let the function $getnc$ determines the neighborhood configuration on a given point for a spatial configuration that involves two volume objects A and B. Then we call $nc_1 \to \ldots \to nc_n$ the neighborhood configuration transition on L from p to q if, and only if, it satisfies the following conditions:*

(i) $nc_1 = getnc(p), nc_n = getnc(q)$

(ii) $\forall\, 1 \leq i < n : nc_i \neq nc_{i+1}$;

(iii) $\exists r_1, \ldots, r_n \in [0, 1]$:

(a) $\forall\, 1 \leq i < n : r_i < r_{i+1}$;

(b) $\forall\, 1 \leq i \leq n : getnc(f(r_i)) = nc_i$;

(c) $\forall\, r_i < r < r_{i+1}, 1 \leq i < n : getnc(f(r)) = getnc(f(r_i)) \vee$
$getnc(f(r)) = getnc(f(r_{i+1}))$;

Definition 31 gives a description about the transitions between the neighborhood configurations of the points along a path in the 3D space. A neighborhood configuration transition describes how the neighborhood configurations of the points change along a path. Different paths between two points may yield different neighborhood configuration transitions. For example, in Figure 3-22A, p, q are two points belonging to A and B respectively where the neighborhood configuration on p is NC_2 and the neighborhood configuration on q is NC_1. Two paths that connect p, q are L_1 and L_2. The neighborhood configuration transition from p to q along path L_1 is $NC_2 \to NC_5 \to NC_1$, while the neighborhood configuration transition from p to q along path L_2 is $NC_2 \to NC_9 \to NC_4 \to NC_{10} \to NC_6 \to NC_1$. The neighborhood configuration transition on L_1 (L_2) describes the changes among neighborhood configurations of the points on L_1 (L_2).

Since the boundary of both simple volumes A and B is path connected, which means that given any two points p, q on the boundary of V with $V \in \{A, B\}$, there

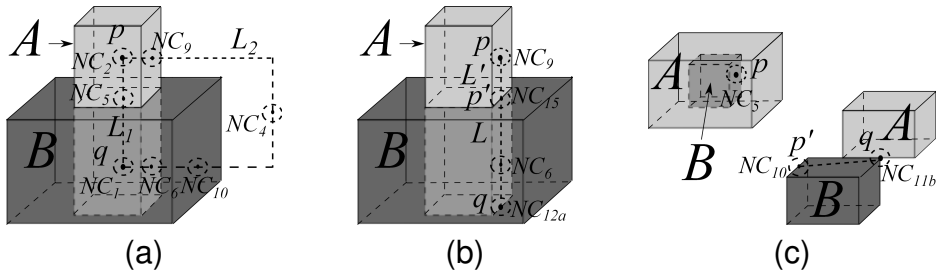


Figure 3-22. Examples of the neighborhood configuration transitions between point p and q.

always exists a path L that connects p and q such that $L \subset \partial V$. Further, if p is on the boundary of A ($p \in \partial A$), the neighborhood configuration on p can only be one of the following neighborhood configurations: NC_6, NC_7, NC_8, NC_9, NC_{11a}, NC_{11b}, NC_{12a}, NC_{12b}, NC_{13a}, NC_{13b}, NC_{14a}, NC_{14b}, and NC_{15}. We call these neighborhood configurations the boundary neighborhood configurations for A. Let BNC_A be a set that collects all boundary neighborhood configurations for A, then BNC_A=$\{NC_6, NC_7, NC_8, NC_9, NC_{11a}, NC_{11b}, NC_{12a}, NC_{12b}, NC_{13a}, NC_{13b}, NC_{14a}, NC_{14b}, NC_{15}\}$. Similarly, the collection of boundary neighborhood configurations for B is BNC_B = $\{NC_5, NC_7, NC_8, NC_{10}, NC_{11a}, NC_{11b}, NC_{12a}, NC_{12b}, NC_{13a}, NC_{13b}, NC_{14a}, NC_{14b}, NC_{15}\}$. Therefore, we call the neighborhood configuration transition on path L from p to q the *boundary neighborhood configuration transition* if $L \subset \partial V$ and $p, q \in \partial V$. A *boundary neighborhood configuration transition* only consists of boundary neighborhood configurations, that is if $nc_1 \rightarrow \ldots \rightarrow nc_n$ is the *boundary neighborhood configuration transition* on path L from p to q ($L \subset \partial V, p \in \partial V, q \in \partial V, V \in \{A, B\}$), then $\forall\, 1 \leq i \leq n$ we have $nc_i \in BNC_V$. For example, in Figure 3-22B $NC_9 \rightarrow NC_{13} \rightarrow NC_6 \rightarrow NC_{12a}$ describes a boundary neighborhood configuration transition on L from p to q where L connects p, q and lies on the boundary of A. All the four neighborhood configurations NC_9, NC_{13}, NC_6 and NC_{12a} belong to the set of boundary neighborhood configurations for A (BNC_A).

Further, let $nc_1 \rightarrow \ldots \rightarrow nc_n$ be a (boundary) neighborhood configuration transition, then we call $nc_i \rightarrow nc_{i+1}$ a direct (boundary) transition from nc_i to nc_{i+1}. According

to Definition 31, a direct transition implies that there exists a path on which only neighborhood configurations nc_1 and nc_2 exist. A neighborhood configuration transition is formed by a sequence of direct neighborhood configuration transitions. When $n = 2$ in a neighborhood configuration transition, then the neighborhood configuration transition itself is a direct transition. For example, the transition $NC_9 \rightarrow NC_{15}$ in Figure 3-22B is a direct boundary neighborhood configuration transition on L' from p to p', which implies that the change of the neighborhood configuration only happened once from NC_9 to NC_{15}, and it is the first part of the boundary neighborhood configuration transition $NC_9 \rightarrow NC_{13} \rightarrow NC_6 \rightarrow NC_{12a}$.

However, we observe that the direct transition does not exist between any two arbitrary neighborhood configurations. In other words, for certain pairs of neighborhood configurations, direct transition between them is not possible. For example, a path that contains direction transition from NC_9 to NC_{12a} does not exist for any spatial configurations. It always involves intermediate neighborhood configurations. In Figure 3-22B, any path lying on the boundary of A that connects two points with their neighborhood configurations as NC_9 and NC_{12a} must first pass through points with neighborhood configurations as NC_{15} and then pass through points with neighborhood configurations as NC_6. No path exists that represents a direct transition $NC_9 \rightarrow NC_{12a}$.

Our last constraint for filtering invalid topological encodings between two simple volumes is based on the requirement of the existence of a neighborhood configuration transition between any two points on the boundary of A and the boundary of B. Due to the boundary path connectedness property for simple volumes, given any two points p, q on the boundary of a simple volume object V, there always exists a path L that connects the two points and lies on the boundary ($L \subset \partial V$). As a result, for any two neighborhood configurations N_x and N_y that exist on the boundary of V, there always exists a neighborhood configuration transition that starts with N_x and ends with N_y ($N_x \rightarrow \ldots \rightarrow N_y$). Therefore, given a topological encoding that identifies all existing

neighborhood configurations in a spatial scenario, it is a valid encoding for two simple volumes A and B if a boundary neighborhood configuration transition exist for any neighborhood configuration pair that exists on the boundary of A or B. Otherwise, the encoding is an invalid one for two simple volumes. For this purpose, we need to first identify all possible direct transitions that exist between boundary neighborhood configuration pairs. Later, these direct transitions will be used to find out all possible boundary neighborhood configuration transitions on the boundary of A and B. We identify these direct transitions between boundary neighborhood configurations in Lemma 10.

Lemma 10. *Let A, B be two volumes in the 3D space, let BNC_A and BNC_B denote the two sets of boundary neighborhood configurations for A and B respectively, and let $directT$ be the function that checks whether a direct transition between two neighborhood configurations is possible. It returns 1 if a direct transition is possible, returns 0 if such a direct transition does not exist. Then, for any two neighborhood configurations NC_x and NC_y on the boundary of A ($NC_x, NC_y \in BNC_A$), we have:*

$$\text{If } x, y \in \{6, 7, 8, 9\} : directT(NC_x, NC_y) = 0;$$

$$\text{If } x, y \in \{11a, 11b, 12a, 12b, 13a, 13b, 14a, 14b\} : directT(NC_x, NC_y) = 0;$$

$$\text{If } x \in \{6, 7, 8, 9, 15\}, y \in \{11a, 11b, 12a, 12b, 13a, 13b, 14a, 14b, 15\} :$$

$$\text{The value of } directT(NC_x, NC_y) \text{ is given in Figure 3-23A;}$$

For any two neighborhood configurations NC_x and NC_y on the boundary of A ($NC_x, NC_y \in BNC_B$), we have:

$$\text{If } x, y \in \{5, 7, 8, 10\} : directT(NC_x, NC_y) = 0;$$

$$\text{If } x, y \in \{11a, 11b, 12a, 12b, 13a, 13b, 14a, 14b\} : directT(NC_x, NC_y) = 0;$$

$$\text{If } x \in \{5, 7, 8, 10\}, y \in \{11a, 11b, 12a, 12b, 13a, 13b, 14a, 14b, 15\} :$$

$$\text{The value of } directT(NC_x, NC_y) \text{ is given in Figure 3-23B;}$$

Proof. For this lemma, we prove the direct transitions for neighborhood configurations on the boundary of A, then the prove for the direct transitions for the neighborhood

configurations on the boundary of B can be solved in a similar manner. We first prove that if $x, y \in \{6, 7, 8, 9\}$, $directT(NC_x, NC_y) = 0$. Let $x = 6$ and $y = 7$, if we assume that a direct transition is possible between NC_6 and NC_7, then there must exist a path L that connects two points p, q whose neighborhood configurations are NC_6 and NC_7 respectively such that $L \subset \partial A$ and the neighborhood configuration of any point on L is either NC_6 or NC_7. As a result, there must exist two points $p, q \in L$ such that (i) the neighborhood configuration of p is NC_6 and the neighborhood configuration of q is NC_7; (ii) p, q are adjacent points on L. According to condition (ii), p is in the neighborhood of q and q is in the neighborhood of p. In other words, in the neighborhood configuration of p (NC_6), there must exist a point q whose neighborhood configuration is NC_7. However, it is not possible since in NC_6 there does not exist points that belong to both the exterior of A and B, while NC_7 contains such points. In the same way, we can prove that no direct transition exists between any x, y pair when $x, y \in \{6, 7, 8, 9\}$ and any x, y pair when $x, y \in \{11a, 11b, 12a, 12b, 13a, 13b, 14a, 14b\}$. Moreover, all the neighborhood pairs in Figure 3-23A that has $directT$ value as 0 can also be proved in the same way. Now, we prove the correctness of the 1 values in Figure 3-23A. We first consider the neighborhood pair NC_6 and NC_{12a}. According to the definition for NC_{12a} (Definition 30) and Lemma 8 (ii), let p denote the point whose neighborhood configuration is NC_{12a}, then there exists a point q in the neighborhood of p that has the neighborhood configuration NC_6. As a result, the neighborhood configuration transition on the path that connects p and q is a direct transition because no other neighborhood configuration exists between p and q. Similarly, we can prove the existence of the direct transitions between other neighborhood configuration pairs that have $directT$ values as 1s. □

In Lemma 10, we have identified for boundary of A (B) all neighborhood configuration pairs between which a direction transition is possible. Finally, we can construct a boundary neighborhood configuration transition graph (*BNCTG*) for any given

$directT$	NC_6	NC_7	NC_8	NC_9	NC_{15}
NC_{11a}	0	0	1	1	1
NC_{11b}	0	0	0	1	1
NC_{12a}	1	1	0	0	1
NC_{12b}	1	0	0	0	1
NC_{13a}	0	1	0	1	1
NC_{13b}	0	0	0	1	1
NC_{14a}	1	0	1	0	1
NC_{14b}	1	0	0	0	1
NC_{15}	1	0	0	1	0

(A)

$directT$	NC_5	NC_7	NC_8	NC_{10}	NC_{15}
NC_{11a}	0	0	1	1	1
NC_{11b}	0	0	0	1	1
NC_{12a}	0	1	0	1	1
NC_{12b}	0	0	0	1	1
NC_{13a}	1	1	0	0	1
NC_{13b}	1	0	0	0	1
NC_{14a}	1	0	1	0	1
NC_{14b}	1	0	0	0	1
NC_{15}	1	0	0	1	0

(B)

Figure 3-23. The direct transition table for the neighborhood configurations on the boundary of A (A) and the direct transition table for the neighborhood configurations on the boundary of B (B).

topological encoding. In principle, we first identify the existing neighborhood configurations according to the given encoding, and we construct two boundary neighborhood configuration transition graphs, one for neighborhood configurations that are on the boundary of A, named as *BNCTG-A*, and the other for neighborhood configurations that are on the boundary of B, named as *BNCTG-B* . In a *BNCTG-A* (*BNCTG-B*), the nodes represent the neighborhood configurations that exist on the boundary of A (B), and each edge that links two nodes represents the possible existence of a direct transition between the two neighborhood configurations. As a result, the *BNCTG-A* (*BNCTG-B*) constructed presents all possible neighborhood configuration transitions in a spatial scenario with the given topological encoding. We give the definitions for *BNCTG-A* and *BNCTG-B* in Definition 32.

Definition 32. *Let $G = (V, E)$ be a undirected graph where V contains a set of nodes and E contains a number of edges linking the nodes. Let A, B be two volumes in a spatial configuration and let E be the 15-bit topological encoding for A and B. We first identify the existence of the 19 neighborhood configurations (including the variants) according to the 15-bit E. Let F_x denote the flag that indicates the existence of neighborhood*

configuration NC_x *where* $x \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11a, 11b, 12a, 12b, 13a, 13b, 14a, 14b, 15\}$. *We have:*

$$\forall 1 \leq i \leq 10: \quad F_i = E[i];$$

$$\forall 10 < i \leq 15: \quad F_{ib} = E[i];$$

$$\forall i \in \{11, 14\}: \quad F_{ia} = E[8] \wedge E[i];$$

$$\forall i \in \{12, 13\}: \quad F_{ia} = E[7] \wedge E[i];$$

Then we call $G_A = (V_A, E_A)$ *the boundary neighborhood configuration transition graph for* A *(*BNCTG-A*) if:*

$$V_A = \bigcup_{(NC_x \in BNC_A) \wedge (F_x = 1)} NC_x;$$

$$E_A = \bigcup_{(NC_x, NC_y \in V) \wedge (directT(NC_x, NC_y) = 1)} (NC_x, NC_y);$$

and we call $G_B = (V_B, E_B)$ *a boundary neighborhood configuration transition graph for* B *(*BNCTG-B*) if:*

$$V_B = \bigcup_{(NC_x \in BNC_B) \wedge (F_x = 1)} NC_x;$$

$$E_B = \bigcup_{(NC_x, NC_y \in V) \wedge (directT(NC_x, NC_y) = 1)} (NC_x, NC_y);$$

According to Definition 32, we can construct both *BNCTG-A* and *BNCTG-B* for a given topological encoding. For example, given the topological encoding 111111101101001 (Figure 3-24), we can identify the existence of 6 neighborhood configurations NC_6, NC_7, NC_9, NC_{12a}, NC_{12b}, and NC_{15} on the boundary of A and 6 neighborhood configurations NC_5, NC_7, NC_{10}, NC_{12a}, NC_{12b}, and NC_{15} on the boundary of B. According to the values in the direct transition tables, Figure 3-23A and Figure 3-23B, we construct the *BNCTG-A* and *BNCTG-B* for this encoding and present them in Figure 3-24. We observe that although there does not exist direct transition between NC_9 and NC_{12a}, there exist a boundary neighborhood configuration transition $NC_9 \rightarrow NC_{15} \rightarrow NC_6 \rightarrow NC_{12a}$. The description of such a transition is presented in Figure 3-22B between point p and q. Further, according to the graph *BNCTG-A*, this is the only possible boundary transition between NC_9 and NC_{12a}, and no

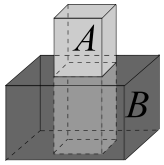
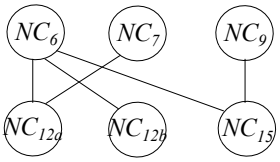
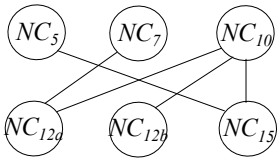
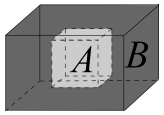
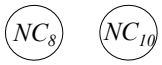
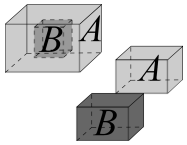
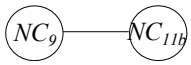
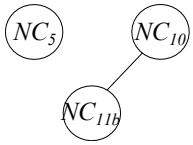
Encoding	BNCTG-A	BNCTG-B
111111101101001 	$V_A = \{NC_6, NC_7, NC_9, NC_{12a}, NC_{12b}, NC_{15}\}$ 	$V_B = \{NC_5, NC_7, NC_{10}, NC_{12a}, NC_{12b}, NC_{15}\}$ 
011100010100000 	$V_A = \{NC_8\}$ 	$V_B = \{NC_8, NC_{10}\}$ 
111110001110000 	$V_A = \{NC_9, NC_{11b}\}$ 	$V_B = \{NC_5, NC_{10}, NC_{11b}\}$ 

Figure 3-24. Examples of the topological encodings with the corresponding BNCTG-A and BNCTG-B.

other transitions between these two neighborhood configurations are possible. Another observation is that, both *BNCTG-A* and *BNCTG-B* for this encoding are connected graph, that is, each neighborhood configuration pair in the graph is connected. In other words, transition is possible between each neighborhood configuration pair. However, this is not the case for all encodings. The second example in Figure 3-24 with the encoding 011100010100000 shows different *BNCTG-A* graph and *BNCTG-B* graph. The spatial configuration described by this encoding consists of a volume B with a cavity and a volume A that fills the cavity of B. Its *BNCTG-A* graph consists only one node, which means that on the boundary of A, only neighborhood configuration NC_8 exists. In its *BNCTG-B* graph, which consists of two isolated nodes NC_8 and NC_{10}, no neighborhood configuration transition on the boundary of B is possible. This means that you can not find a path L that lies on the boundary of B and connects two points with their neighborhood configurations as NC_8 and NC_{10}. The third example shown in Figure 3-24 with the encoding 111110001110000, has a connected *BNCTG-A* graph but a disconnected *BNCTG-B* graph. In its *BNCTG-B* graph, no connection exists

between NC_5 and NC_{10} and no connection exist between NC_5 and NC_{11b}. The example in Figure 3-22C demonstrates such relationships. There does not exit a path lying on the boundary of B that connects point p with q nor such path that connects point p with p'. Only the path that connects p' with q exists that lies entirely on the boundary of B. This is consistent with what *BNCTG-B* graph describes.

Finally, we can present the last constraint for two simple volumes, that is, the encodings that generate either disconnected *BNCTG-A* or disconnected *BNCTG-B* do not represent topological relationships between two simple volume objects. We present this constraint in Lemma 11.

Lemma 11. *Let E be a topological encoding for two simple volumes A and B, then each node pair in its* BNCTG-A *must be connected and each node pair in its* BNCTG-B *must be connected.*

Proof. Assume there exist two nodes n_x and n_y in *BNCTG-A* that are not connected. Let the points p, q denote two points whose neighborhood configurations are n_x and n_y. Then there does not exist a path L that connects p and q such that L lies on the boundary of A. This is a contradiction against the path connectedness property of the boundary of A. Thus the assumption is invalid. Similarly, we can prove that *BNCTG-B* must be connected. □

According to Lemma 11, the second and third encodings in Figure 3-24 do not qualify for valid topological encodings between two simple volumes.

We have developed a program to check all the constraints against the 697 topological encodings that are valid for two complex volumes, and we get a total of 72 encodings survives the constraints. To prove the validity of all the 72 encodings for two simple volumes, we create a spatial configuration for each encoding. We present all 72 spatial configuration drawings together with their corresponding topological relationship encodings and the corresponding 9IM matrices in Figure A-4 to Figure A-6.

Finally, we can conclude that our NCM model is able to distinguish a total of 72 different topological relationships between two simple volume objects in the 3D space.

3.2.1.5 Comparison of the NCM with the 9-Intersection Model

In Section 3.2.1.3 and Section 3.2.1.4, we have evaluated our NCM model in terms of the number of topological relationships that can be distinguished between two complex volumes and between two simple volumes. We have proved that a total of 697 topological relationships can be distinguished by our NCM between two complex volumes and a total of 72 topological relationships can be distinguished between two simple volumes. In this section, we compare our NCM model with the most popular modeling strategy for topological relationships, the 9-intersection matrix model (9IM), and show that for 3D volumes, our NCM is more powerful than the 9IM.

Our neighborhood configuration model and the 9-intersection based models share an important basis, which is point set theory and point set topology. This basis is important because it enables the modeling strategies to stay in the abstract topological space where only point sets are used and the discrete representation of a spatial object is not relevant. As a result, models based on point sets are more general than other models that leave the abstract topological space. No matter what data representation of a spatial object is used, the models based on the point sets will always work. On the other hand, models based on one particular representation of spatial objects may not work or may yields different results on another representation of the same spatial objects. For example, the dimensional model [107] (DM) is based on the dimensional elements of spatial objects, which determines the corner points of a region object as 0D elements. It highly depends on the representation of a spatial object, and it yields different results for a circle with a smooth boundary and a circle with its boundary approximated with a set of connected segments.

Therefore, in this section, we compare our NCM model with the 9IM model, and show that we can distinguish more topological relationships between two volume objects.

First, we show the relationship between the 9IM and our NCM. The 9IM model is based on the nine possible intersections of the boundary (∂A), interior (A°), and exterior (A^-) of a spatial object A with the corresponding components of another object B. A 9 element matrix is used to represent the empty and non-emptiness of the intersections between the components. The topological relationship between two spatial objects A and B can be expressed by evaluating the matrix in Figure 2-1. By comparing the 15 neighborhood configurations with the 9 elements in a 9IM matrix, we observe that there exist correlations between them. Let the function $9IM(A, B)$ yield the 9IM matrix for two spatial objects A and B, and let the function $NCM(A, B)$ yield the topological encoding for A and B, then Lemma 12 shows the correlation between them.

Lemma 12. *Let A, B be two spatial objects, let $M = 9IM(A, B)$ be a matrix with 9 binary elements, and let $E = NCM(A, B)$ be a 15 bit binary encoding, then we have:*

$$M[0][0] \Leftrightarrow E_1 \vee E_5 \vee E_6 \vee E_7 \vee E_{12} \vee E_{13} \vee E_{14} \vee E_{15}$$

$$M[0][1] \Leftrightarrow E_5 \vee E_{13} \vee E_{14} \vee E_{15}$$

$$M[0][2] \Leftrightarrow E_2 \vee E_5 \vee E_8 \vee E_9 \vee E_{11} \vee E_{13} \vee E_{14} \vee E_{15}$$

$$M[1][0] \Leftrightarrow E_6 \vee E_{12} \vee E_{14} \vee E_{15}$$

$$M[1][1] \Leftrightarrow E_7 \vee E_8 \vee E_{11} \vee E_{12} \vee E_{13} \vee E_{14} \vee E_{15}$$

$$M[1][2] \Leftrightarrow E_9 \vee E_{11} \vee E_{13} \vee E_{15}$$

$$M[2][0] \Leftrightarrow E_3 \vee E_6 \vee E_8 \vee E_{10} \vee E_{11} \vee E_{12} \vee E_{14} \vee E_{15}$$

$$M[2][1] \Leftrightarrow E_{10} \vee E_{11} \vee E_{12} \vee E_{15}$$

$$M[2][2] \Leftrightarrow E_4 \vee E_7 \vee E_9 \vee E_{10} \vee E_{11} \vee E_{12} \vee E_{13} \vee E_{15}$$

Proof. Due to space limit, we only prove the first correspondence, and the others can be proved in a similar way. If $M[0][0] = 1$, which indicates that $A^\circ \cap B^\circ \neq \emptyset$, then there exist a point $p \in A^\circ \cap B^\circ$. This means that $p \in A^\circ$ and $p \in B^\circ$, then according to the

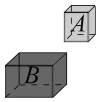
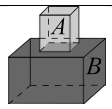
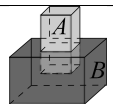
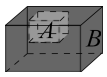
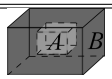
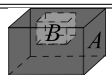
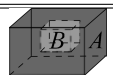
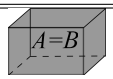
				
9IM	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(011100001100000)	(011100011110000)	(111111001100001)	(101101100101000)
				
9IM	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
NCM	(101101000100000)	(110110101000100)	(110110001000000)	(100100100000000)

Figure 3-25. The 8 topological relationships between two volumes that can be distinguished with both 9IM and NCM.

definitions of the neighborhood configurations, NC_1, NC_5, NC_6, NC_7, NC_{12}, NC_{13}, NC_{14}, and NC_{15} are the neighborhood configurations that contains such point p. Hence, one of these neighborhood configurations must exist. □

According to Lemma 12, given a topological encoding E for two spatial objects A and B, which indicates the existence of the 15 neighborhood configurations, the corresponding 9IM matrix M for A and B can be derived, but not vice versa. For example, given the encoding 111111101101001 (see the first example in Figure 3-24), we can derive a unique 9IM matrix M in which every element has value 1 (see the third example in Figure 3-26). However, if we are given the matrix M with every element having value 1, we will not be able to determine a unique encoding. Several topological encodings are possible for M, e.g., 111111011110011, 111111011110001, 111111101101001, 111111111111001, etc. As a result, more than one topological encodings map to one 9IM matrix, hence, our NCM model is capable of distinguishing more topological relationships.

According to [31], 8 topological relationships are distinguished between two simple 3D volume objects. Figure 3-25 shows the 8 configurations and their corresponding 9IM matrices. To demonstrate that our model can also distinguish these 8 topological relationships, we also list our topological relationship encoding for each configuration

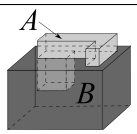
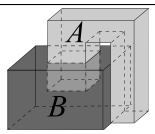
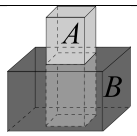
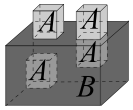
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(111111011110011)	(111111011110001)	(111111101101001)	(111111111111001)

Figure 3-26. The 4 topological relationships that can be distinguished with NCM but not 9IM.

in Figure 3-25. We observe that for all 8 topological relationships the topological relationship encodings are also different.

Moreover, the 9IM model only picks up the dominating topological relationships such as overlap, and ignores other topological relationship factors that may also exist at the same time, e.g. meet at a face. However, since we start with a very small neighborhood of each point, we are able to capture more fine-grained topological relationships. In Figure 3-26, we have listed 4 topological relationships that can be distinguished with our NCM model but are all identified simply as overlap in the 9IM model. Apart from the overlap relationship between A and B, the topological relationships like A meets B on a face from the outside (first and second scenario in Figure 3-26), A meets B on a face from the inside of B (third scenario in Figure 3-26), and A has a component inside B and touches B on a face (fourth scenario in Figure 3-26) are all distinguished in our model.

Therefore, we can conclude that our NCM model is more powerful than 9IM based models in terms of differentiating topological relationships between two volume objects, especially complex volumes.

3.2.2 3D Cardinal Direction Relationships Modeling

Another set of important spatial relationships are called the *cardinal directional relationships* between spatial objects. It has a long research tradition in spatial databases, GIS, and disciplines like cognitive science, robotics, artificial intelligence, and qualitative spatial reasoning. Cardinal directions represent absolute directional

relationships like *north* and *southwest* with respect to a given reference system. In spatial databases, they are usually integrated into query languages as selection and join conditions. So far, most efforts have focused on modeling cardinal directions between objects in the two-dimensional (2D) space while efforts in the 3D space have been rare. Besides the known 2D cardinal directions, the third dimension introduces new direction components such as *above* and *below* that have to be taken into account in 3D cardinal direction models. We have reviewed the existing 3D models for cardinal directions in Section 2.4.2, most of whom suffer from one of the following problems. First, they do not consider the shapes of the 3D objects and either approximate these objects as single 3D points or as minimum bounding cubes. Hence, they loose precision and sometimes lead to incorrect results. Second, in some models the two 3D operand objects are treated in an unequal manner, thus leads to the violation of the principle of converseness. That is, the cardinal direction determined for two 3D spatial objects A and B is not always equal to the inverse of the cardinal relation determined for B and A. Third, some models do not have a complete coverage of all possible relations. Thus, in Section 3.2.2.1, we propose a new two-phase model, called object interaction cube matrix (*OICM*) model that solves these problems.

Further, as a detour from the 3D problems, we investigate the representation of cardinal direction development between two moving objects. Recently, a wide range of applications like hurricane research, fire management, navigation systems, and transportation, to name only a few, has shown increasing interest in managing and analyzing space and time-referenced objects, so-called moving objects, that continuously change their positions over time. In the same way as moving objects can change their location over time, the cardinal directions between them can change over time. Transferred to a spatiotemporal context, the simultaneous location change of different moving objects can imply a temporal evolution of their directional relationships, called development. It is an open, interesting, and challenging problem to capture

the cardinal direction development between moving objects. Thus, from a modeling perspective, we defined the development of cardinal directions over time as a sequence of temporally ordered and enduring cardinal directions. We present the modeling strategy in Section 3.2.2.2.

3.2.2.1 The Objects Interaction Cube Matrix for 3D Complex Volumes

The main idea of our novel Objects Interaction Cube Matrix (*OICM*) model is to capture and represent the interactions between two general 3D volumes A and B with possibly disconnected components and cavities and to derive the cardinal direction from this information. Our model consists of a representation phase and an interpretation phase. In the representation phase, we create an $k \times m \times n$-*Objects Interaction Cube* (*OIC*) ($k, m, n \in \{1, 2, 3\}$) by intersecting their minimum bounding boxes with each other to capture all possible interactions between two 3D volume objects. We use an *Objects Interaction Cube Matrix* (*OICM*) to keep which object intersects which sub-cube. In the interpretation phase, we develop a technique to compute the cardinal direction from the matrix and define the semantic of the result obtained. Each phase will be detailed in the rest of this section.

- Representing interactions of objects in 3D Space with the Objects Interaction Cube Matrix (*OICM*)

The general idea is to first superimpose a cube called objects interaction cube on a configuration of two 3D spatial objects (volumes) to capture their interactions. Such a cube is constructed from a total of twelve partitioning planes derived from both objects. Partitioning planes are the infinite extensions of the faces of the axis-aligned minimum bounding box around each object; each partitioning plane is perpendicular to one particular coordinate axis. The six partitioning planes from each object create a partition of the Euclidean space $\mathbb{R}^3$ into 27 mutually exclusive cells from which only the central cell is bounded and the other 26 cells are unbounded. The object itself is located in the bounded, central cell. This essentially describes the tiling strategy of the TCD model.

However, our fundamental improvement is that we employ this tilling strategy to both objects, thus obtain two separate space partitions, and then overlay both partitions. The overlay creates a new subdivision of space, and further, also partitions each object into non-overlapping components, where each component of one object lies in a different bounded cell. This overlay achieves coequal interaction and symmetric treatment of the two objects. In the most general case, all partitioning planes are different from each other, and the overlay creates 27 non-overlapping bounded cells and several unbounded cells. Since the components of the objects are always located inside the bounded cells, we can exclude all unbounded cells and only focus on the bounded cells that are relevant to the objects' interactions. Definition 33 formally defines the objects interaction cube space that contains all bounded cells.

Definition 33. *Let* $A, B \in$ *volume with* $A \neq \varnothing$ *and* $B \neq \varnothing$*, and let* $min_x^v = \min\{x \,|\, (x,y,z) \in v\}$*,* $max_x^v = \max\{x \,|\, (x,y,z) \in v\}$*,* $min_y^v = \min\{y \,|\, (x,y,z) \in v\}$*,* $max_y^v = \max\{y \,|\, (x,y,z) \in v\}$*,* $min_z^v = \min\{z \,|\, (x,y,z) \in v\}$*, and* $max_z^v = \max\{z \,|\, (x,y,z) \in v\}$ *for* $v \in \{A, B\}$*. Then the objects interaction cube space* (OICS) *of* A *and* B *is given as*

$$\begin{aligned}\mathrm{OICS}(A,B) = \{(x,y,z) \in \mathbb{R}^3 \,|\quad & \min(min_x^A, min_x^B) \leq x \leq \max(max_x^A, max_x^B) \wedge \\ & \min(min_y^A, min_y^B) \leq y \leq \max(max_y^A, max_y^B) \wedge \\ & \min(min_z^A, min_z^B) \leq z \leq \max(max_z^A, max_z^B)\}\end{aligned}$$

Definition 34 defines partitioning faces as part of the partitioning plane and superimposes them on the objects interaction cube space to obtain the objects interaction cube.

Definition 34. *Let* $f_{min\_x}^v$*,* $f_{max\_x}^v$*,* $f_{min\_y}^v$*,* $f_{max\_y}^v$*,* $f_{min\_z}^v$*, and* $f_{max\_z}^v$ *denote the six partitioning faces of* v *for* $v \in \{A, B\}$*, then*

$$f^v_{min\_x} = \{(x, y, z) \in OICS(A, B) \,|\, x = min^v_x\}$$
$$f^v_{max\_x} = \{(x, y, z) \in OICS(A, B) \,|\, x = max^v_x\}$$
$$f^v_{min\_y} = \{(x, y, z) \in OICS(A, B) \,|\, y = min^v_y\}$$
$$f^v_{max\_y} = \{(x, y, z) \in OICS(A, B) \,|\, y = max^v_y\}$$
$$f^v_{min\_z} = \{(x, y, z) \in OICS(A, B) \,|\, z = min^v_z\}$$
$$f^v_{max\_z} = \{(x, y, z) \in OICS(A, B) \,|\, z = max^v_z\}$$

Next, let F_x, F_y and F_z denote the three sets that contain the partitioning faces perpendicular to the x-axis, the y-axis and the z-axis respectively. We obtain

$$F_x = \{f^A_{min\_x}, f^A_{max\_x}, f^B_{min\_x}, f^B_{max\_x}\} \quad F_y = \{f^A_{min\_y}, f^A_{max\_y}, f^B_{min\_y}, f^B_{max\_y}\}$$
$$F_z = \{f^A_{min\_z}, f^A_{max\_z}, f^B_{min\_z}, f^B_{max\_z}\}$$

*Finally, we get the ("cell walls" of the) objects interaction cube (*OIC*) for A and B (see Figures 3-27A and 3-27B) as*

$$OIC(A, B) = F_x \cup F_y \cup F_z$$

This definition comprises all cases for OICs. In general, we obtain a $3 \times 3 \times 3$ OIC if $|F_x| = |F_y| = |F_z| = 4$, meaning that there are a total of 12 different partitioning faces that forms 27 bounded cells. Special cases arise if at least one of the three sets F_x, F_y, and F_z contains less than four partitioning faces; this means that at least two partitioning faces from the two objects coincide. This results in different sizes of cubes. Definition 35 formally describes the situation.

Definition 35. *An objects interaction cube* OIC(A, B) *is of size $k \times m \times n$, with $k, m, n \in \{1, 2, 3\}$, if $|F_x| = k + 1$, $|F_y| = m + 1$, and $|F_z| = n + 1$.*

The objects interaction cube space is partitioned into several cubic cells by the partitioning faces (see Definition 36).

Definition 36. *Since each partitioning face is perpendicular to a specific coordinate axis, we first define an ordering of the partitioning planes that are perpendicular to the same axis. Let $h_1, h_2 \in F_r$ with $r \in \{x, y, z\}$, then $h_1 < h_2$ if $p_1.r < p_2.r$ for any $p_1 \in h_1$ and any*

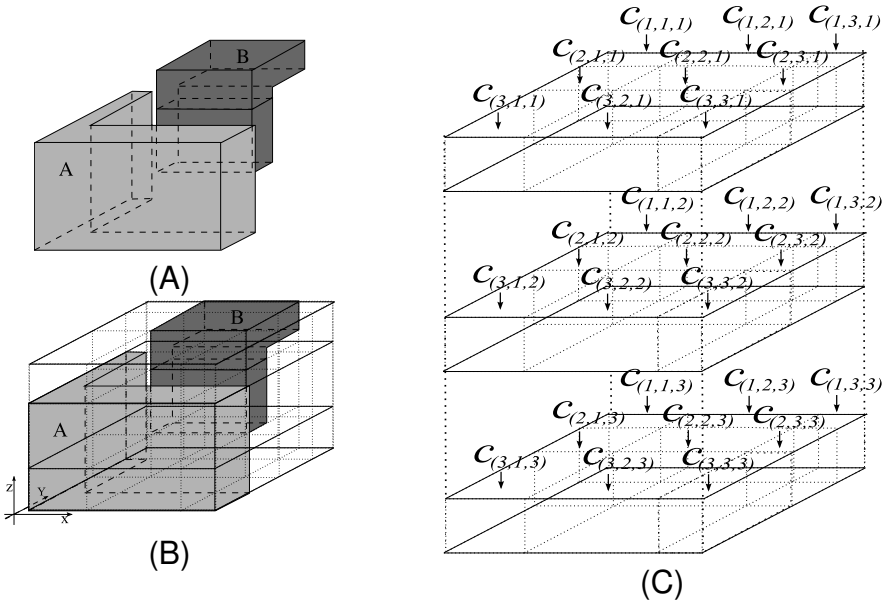


Figure 3-27. The OICM model for A and B. Examples of two volumes A and B (A), their OIC representation (B), and the labeling of the OIC cubic cells (C).

$p_2 \in h_2$. *Next, we define three auxiliary lists that store partitioning faces regarding the order* $<$*:*

$$list_{left\_right} = \{\langle f_1, \ldots, f_{k'}\rangle \mid k' = |F_x|, \forall 1 \le i < j \le k' \; \forall f_i, f_j \in F_x : f_i < f_j\}$$

$$list_{behind\_front} = \{\langle g_1, \ldots, g_{m'}\rangle \mid m' = |F_y|, \forall 1 \le i < j \le m' \; \forall g_i, g_j \in F_y : g_i > g_j\}$$

$$list_{top\_down} = \{\langle h_1, \ldots, h_{n'}\rangle \mid n' = |F_z|, \forall 1 \le i < j \le n' \; \forall h_i, h_j \in F_z : h_i > h_j\}$$

Further, for the partitioning faces $f \in F_r$ *with* $r \in \{x, y, z\}$*, let the function* $get_r(f)$ *return the common value* r *of all points of the face* f*. An objects interaction cubic cell* $c_{(i,j,l)}$ *with* $1 \le i \le k$*,* $1 \le j \le m$ *and* $1 \le l \le n$ *is then defined as:*

$$c_{(i,j,l)} = \{(x, y, z) \in \text{OIGS}(A, B) \mid get_x(f_i) \le x \le get_x(f_{i+1}),$$
$$get_y(g_j) \le y \le get_y(g_{j+1}),$$
$$get_z(h_l) \le z \le get_z(h_{l+1})\}$$

Figure 3-27C shows the visualization of the labeling of cubic cells in a $3 \times 3 \times 3$ OIC.

The objects interaction cube for two volume objects A and B provides us with the valuable information which volume object intersects with cubic cell. Definition 37 provides the definition of the interaction between A, B, and a cubic cell.

Definition 37. *Given $A, B \in$ volume with $A \neq \varnothing$ and $B \neq \varnothing$, let ι be a function that encodes the interaction of A and B with a cubic cell $c_{(i,j,l)}$, and checks whether no object, A only, B only, or both objects intersect a cubic cell. We define this function as*

$$\iota(A, B, c_{(i,j,l)}) = \begin{cases} 0 \textit{ if} & A^\circ \cap c^\circ_{(i,j,l)} = \varnothing \ \wedge \ B^\circ \cap c^\circ_{(i,j,l)} = \varnothing \\ 1 \textit{ if} & A^\circ \cap c^\circ_{(i,j,l)} \neq \varnothing \ \wedge \ B^\circ \cap c^\circ_{(i,j,l)} = \varnothing \\ 2 \textit{ if} & A^\circ \cap c^\circ_{(i,j,l)} = \varnothing \ \wedge \ B^\circ \cap c^\circ_{(i,j,l)} \neq \varnothing \\ 3 \textit{ if} & A^\circ \cap c^\circ_{(i,j,l)} \neq \varnothing \ \wedge \ B^\circ \cap c^\circ_{(i,j,l)} \neq \varnothing \end{cases}$$

The operator $^\circ$ denotes the point-set topological interior operator and yields a volume without its boundary. The function ι encodes the interaction between two volumes and a cubic cell of their OIC. In order to represent a three dimensional $k \times m \times n$ cube with a two dimensional matrix, we introduce a cubic column vector $v_{i,j} = \langle c_{(i,j,1)}, \ldots, c_{(i,j,n)} \rangle$ with $1 \leq i \leq k$ and $1 \leq j \leq m$. It represents a list of cubic cells that are ordered along the z-axis in a top-down fashion. Further, we define a variant of the function ι, ι', that takes the objects A, B, and the cubic column vector $v_{i,j}$ as inputs, and returns a vector of encodings. So we have $\iota'(A, B, v_{i,j}) = \langle \iota(A, B, c_{(i,j,1)}), \ldots, \iota(A, B, c_{(i,j,n)}) \rangle$. Thus for each cubic column vector, we store the encoded interaction information in a vector element of an objects interaction cube matrix (*OICM*). In this way, we establish a mapping from the objects interaction cube, which is the geometric representation of the objects' interaction, to the matrix representation of the objects' interaction information. We obtain 27 possible objects interaction cubes of size $k \times m \times n$ since $k, m, n \in \{1, 2, 3\}$. These are mapped to 27 possible objects interaction cube matrices of size $k \times m$ with vector element length n. As an example, we show the objects interaction cube matrix $OICM(A, B)$ for a given $3 \times 3 \times 3$ objects interaction cube $OIC(A, B)$:

$$OICM(A, B) = \begin{pmatrix} \iota'(A, B, v_{1,1}) & \iota'(A, B, v_{1,2}) & \iota'(A, B, v_{1,3}) \\ \iota'(A, B, v_{2,1}) & \iota'(A, B, v_{2,2}) & \iota'(A, B, v_{2,3}) \\ \iota'(A, B, v_{3,1}) & \iota'(A, B, v_{3,2}) & \iota'(A, B, v_{3,3}) \end{pmatrix}$$

$$\begin{pmatrix} \langle 0,0,0\rangle & \langle 2,0,0\rangle & \langle 0,0,0\rangle \\ \langle 0,1,1\rangle & \langle 2,3,1\rangle & \langle 0,1,1\rangle \\ \langle 0,1,1\rangle & \langle 0,1,1\rangle & \langle 0,1,1\rangle \end{pmatrix} \quad \begin{pmatrix} \langle 0,0,0\rangle & \langle 2,0,0\rangle & \langle 0,0,0\rangle \\ \langle 0,1,1\rangle & \langle 2,2,0\rangle & \langle 0,0,0\rangle \\ \langle 0,1,1\rangle & \langle 0,1,1\rangle & \langle 0,1,1\rangle \end{pmatrix}$$

(A) (B)

Figure 3-28. The OICM for the previous example. The OICM for the example in Figure 2-3B (A), and the OICM for the example in Figure 3-27A (B).

The elements in the matrix are vectors that store the coded interaction information. Figure 3-28 shows two matrices computed from the examples in Figure 2-3B and Figure 3-27A.

- Interpreting direction relations with the OICM model

The second phase of the OICM model is the interpretation phase. It takes an objects interaction cube matrix obtained as the result of the representation phase as input and uses it to generate a set of cardinal directions as output. A new cardinal direction set with 27 basic cardinal directions is first defined. Then the interpretation phase based on an OICM is detailed.

As we have seen, for any two complex volumes, an OIC with $k \times m \times n$ non-overlapping cubic cells can be generated, where $k, m, n \in \{1, 2, 3\}$. Any cubic cell $c_{(i,j,l)}$ ($1 \leq i \leq k, 1 \leq j \leq m, 1 \leq l \leq n$) can be located by its index triplet (i, j, l). Taking the triplets of two cubic cells, we aim at deriving their cardinal direction. For this, we need a cardinal direction model. We could use a popular model in the 2D space with the nine cardinal directions *north* (*N*), *northwest* (*NW*), *west* (*W*), *southwest* (*SW*), *south* (*S*), *southeast* (*SE*), *east* (*E*), *northeast* (*NE*), and *origin* (*O*) to denote the possible cardinal directions between cubic cells. We call the elements of the set $CD_{2D} = \{N, NW, W, SW, S, SE, E, NE, O\}$ *basic 2D cardinal directions*. However, this model ignores the different heights of spatial objects towards each other. The height is represented by the z-dimension and in addition enables us to distinguish whether an object is located upper (u), at the same level (s), or *lower* (l) than another object. Taken together, we obtain cardinal directions like SE_u, SE_s and SE_l which have the meaning of

Table 3-3. Interpretation table for the interpretation function ψ

ψ	condition	ψ	condition	ψ	condition
NW_u	$x_1 \wedge y_1 \wedge z_1$	NW_s	$x_1 \wedge y_1 \wedge z_2$	NW_l	$x_1 \wedge y_1 \wedge z_3$
N_u	$x_1 \wedge y_2 \wedge z_1$	N_s	$x_1 \wedge y_2 \wedge z_2$	N_l	$x_1 \wedge y_2 \wedge z_3$
NE_u	$x_1 \wedge y_3 \wedge z_1$	NE_s	$x_1 \wedge y_3 \wedge z_2$	NE_l	$x_1 \wedge y_3 \wedge z_3$
W_u	$x_2 \wedge y_1 \wedge z_1$	W_s	$x_2 \wedge y_1 \wedge z_2$	W_l	$x_2 \wedge y_1 \wedge z_3$
O_u	$x_2 \wedge y_2 \wedge z_1$	O_s	$x_2 \wedge y_2 \wedge z_2$	O_l	$x_2 \wedge y_2 \wedge z_3$
E_u	$x_2 \wedge y_3 \wedge z_1$	E_s	$x_2 \wedge y_3 \wedge z_2$	E_l	$x_2 \wedge y_3 \wedge z_3$
SW_u	$x_3 \wedge y_1 \wedge z_1$	SW_s	$x_3 \wedge y_1 \wedge z_2$	SW_l	$x_3 \wedge y_1 \wedge z_3$
S_u	$x_3 \wedge y_2 \wedge z_1$	S_s	$x_3 \wedge y_2 \wedge z_2$	S_l	$x_3 \wedge y_2 \wedge z_3$
SE_u	$x_3 \wedge y_3 \wedge z_1$	SE_s	$x_3 \wedge y_3 \wedge z_2$	SE_l	$x_3 \wedge y_3 \wedge z_3$

upper southeast, same level southeast, and lower southeast, respectively. In general, we obtain a refined version of CD_{2D} into the set $CD_{3D} = \{N_u, N_s, N_l, NW_u, NW_s, NW_l, \ldots, NE_u, NE_s, NE_l, O_u, O_s, O_l\}$ of 27 jointly exhaustive and pairwise disjoint basic 3D cardinal directions between two cubic cells in an objects interaction cube. A different underlying set of basic cardinal directions would lead to a different interpretation of cubic cell pairs. Definition 38 gives a specification of the basic cardinal directions in terms of an interpretation function.

Definition 38. *Let $A, B \in$ volume with $A \neq \varnothing$ and $B \neq \varnothing$, and let $c_{(i_1,j_1,l_1)}$ and $c_{(i_2,j_2,l_2)}$ denote the two cubic cells in the $k \times m \times n$ objects interaction cube $OIC(A, B)$ with $k, m, n \in \{1, 2, 3\}$, $1 \leq i_1, i_2 \leq k, 1 \leq j_1, j_2 \leq m, 1 \leq l_1, l_2 \leq n$. Then we define the following equivalences:*

$$x_1 :\Leftrightarrow i_1 > i_2 \qquad x_2 :\Leftrightarrow i_1 = i_2 \qquad x_3 :\Leftrightarrow i_1 < i_2$$

$$y_1 :\Leftrightarrow j_1 > j_2 \qquad y_2 :\Leftrightarrow j_1 = j_2 \qquad y_3 :\Leftrightarrow j_1 < j_2$$

$$z_1 :\Leftrightarrow l_1 > l_2 \qquad z_2 :\Leftrightarrow l_1 = l_2 \qquad z_3 :\Leftrightarrow l_1 < l_2$$

Further, let $\psi((i_1, j_1, l_1), (i_2, j_2, l_2))$ denote the interpretation function that takes the location index triplets (i_1, j_1, l_1) and (i_2, j_2, l_2) of two cubic cells as input and yields the cardinal direction from cubic cell $c_{(i_1,j_1,l_1)}$ to $c_{(i_2,j_2,l_2)}$. Then Table 3-3 provides the definition of the interpretation function.

In Definition 38, a total of 27 basic cardinal directions between two cubic cells in 3D space are defined. For any given two cubic cells in the objects interaction cube,

their directional relationship d belongs to the set CD_{3D} ($d \in CD_{3D}$). For example, in Figure 3-27C, the cardinal direction from cubic cell $c_{(2,2,1)}$ to $c_{(3,1,2)}$ is by definition $\psi((2,2,1),(3,1,2)) = SW_l$, which can be expressed as the cubic cell $c_{(3,1,2)}$ is to the lower southwest of the cubic cell $c_{(2,2,1)}$.

The objects interaction cube subdivides each volume object into a set of non-overlapping components that are located in different cubic cells. As a result, the cardinal direction between any two components is equivalent to the cardinal direction between the two cubic cells that hold them. This equivalence ensures the correctness of applying the set CD_{3D}, which is for two cubic cells, to symbolize the possible cardinal directions between object components.

As a first step of the interpretation phase, we define a function loc (see Definition 39) that acts on one of the volume objects A or B and their common objects interaction cube matrix and determines all locations of components of each object in the matrix. Let M denote an objects interaction matrix, then we use an index triplet (i, j, l) to represent the location of the lth element in the vector element $M_{i,j}$ and thus the location of an object component in the objects interaction cube matrix M.

Definition 39. *Let M be the $k \times m$-objects interaction cube matrix of two volume objects A and B with the vector element length n. Then the function loc is defined as:*

$$loc(A, M) = \{(i, j, l) \mid 1 \leq i \leq k, 1 \leq j \leq m, 1 \leq l \leq nM_{i,j}[l] = 1 \ \vee \ M_{i,j}[l] = 3\}$$

$$loc(B, M) = \{(i, j, l) \mid 1 \leq i \leq k, 1 \leq j \leq m, 1 \leq l \leq nM_{i,j}[l] = 2 \ \vee \ M_{i,j}[l] = 3\}$$

Once the location of a component is computed, we can apply the interpretation function ψ, which takes the location index triplets of two components and produces the cardinal direction between them.

Finally, we specify the cardinal direction function named *dir* which determines the composite cardinal direction for two volume objects A and B. This function has the signature $dir : volume \times volume \rightarrow 2^{CD_{3D}}$ and yields a set of basic cardinal directions as its result. We now specify the cardinal direction function dir in Definition 40.

Definition 40. *Let $A, B \in$ volume. Then the* cardinal direction function dir *is defined as*

$$dir(A,B) = \{\psi((i,j,l),(i',j',l')) \mid \quad (i,j,l) \in Ioc(A, OICM(A,B)),$$
$$(i',j',l') \in Ioc(B, OICM(A,B))\}.$$

Function *dir* yields the union of the basic cardinal directions between all component pairs of both objects. It determines all cardinal directions that exist between two volume objects. To illustrate the interpretation phase, we use our example in Figure 3-28B. From the objects interaction cube matrix $OICM(A,B)$, we can obtain $Ioc(A, OICM(A,B))$ and $Ioc(B, OICM(A,B))$ as follows:

$$Ioc(A, OICM(A,B)) = \{(2,1,2),(2,1,3),(3,1,2),$$
$$(3,1,3),(3,2,2),(3,2,3),(3,3,2),(3,3,3)\}$$
$$Ioc(B, OICM(A,B)) = \{(1,2,1),(2,2,1),(2,2,2)\}$$

The cardinal directions between the two volume objects A and B can then be derived with the cardinal direction function *dir*.

$$dir(A,B) = \{\psi((2,1,2),(1,2,1)), \psi((2,1,3),(1,2,1)), \ldots, \psi((3,3,3),(2,2,2))\}$$
$$= \{N_s, N_u, E_s, E_u, NE_s, NE_u, NW_s, NW_u\}$$

Similarly, we obtain the inverse cardinal direction as:

$$dir(B,A) = \{\psi((1,2,1),(2,1,2)), \psi((2,2,1),(2,1,2)), \ldots, \psi((2,2,1),(3,3,3))\}$$
$$= \{S_s, S_l, W_s, W_l, SW_s, SW_l, SE_s, SE_l\}$$

Finally we can say regarding Figure 3-28B that "Object B is partly same level north, partly upper north, partly same level east, partly upper east, partly same level northeast, partly upper northeast, partly same level northwest, and partly upper northwest of object A" and that "Object A is partly same level south, partly lower south, partly same level west, partly lower west, partly same level southwest, partly lower southwest, partly same level southeast, and partly lower southeast of object B", which is consistent.

3.2.2.2 Detour: Modeling the Cardinal Direction Development between Moving Points

We have discussed the cardinal directions between two static spatial objects. Transferred to a spatio-temporal context, the simultaneous location change of different moving objects can imply a change of their directional relationships. For example, a

fishing boat that is southwest of a storm might be north of it some time later. We call this a cardinal direction development. Such a development between two moving objects describes a temporally ordered sequence of cardinal directions where each cardinal direction holds for a certain time interval during their movements. A development reflects the impact of time on the directional relationships between two moving objects, and usually proceeds continuously over time if the movements of the two objects are continuous. In this section, we formally define the cardinal direction development between two moving points. We start with a review of the cardinal directions between two static points.

- Cardinal directions between static points

The approach that is usually taken for defining cardinal directions between two static points in the Euclidean plane is to divide the plane into partitions using the two points. One popular partition method is the projection-based method that uses lines orthogonal to the x- and y-coordinate axes to make partitions [40, 78]. The point that is used to create the partitions is called the reference point, and the other point is called the target point. The directional relation between two points is then determined by the partition that the target object is in, with respect to the reference object. Let $Points$ denote the set of static point objects, and let $p, q \in Points$ be two static point objects, where p is the target point and q is the reference point. A total of 9 mutually exclusive cardinal directions are possible between p and q. Let CD denote the set of 9 cardinal directions, then CD=$\{$*northwest* (NW), *restrictednorth* (N), *northeast* (NE), *restrictedwest* (W), *sameposition*(SP), *restrictedeast* (E), *southwest* (SW), *restrictedsouth* (S), *southeast* (SE)$\}$. Further, let X and Y be functions that return the x and y coordinate of a point object respectively. The cardinal direction $dir(p, q) \in CD$ between p and q is therefore defined as

$$dir(p, q) = \begin{cases} NW & \text{if } X(p) < X(q) \wedge Y(p) > Y(q) \\ N & \text{if } X(p) = X(q) \wedge Y(p) > Y(q) \\ NE & \text{if } X(p) > X(q) \wedge Y(p) > Y(q) \\ W & \text{if } X(p) < X(q) \wedge Y(p) = Y(q) \\ SP & \text{if } X(p) = X(q) \wedge Y(p) = Y(q) \\ E & \text{if } X(p) > X(q) \wedge Y(p) = Y(q) \\ SW & \text{if } X(p) < X(q) \wedge Y(p) < Y(q) \\ S & \text{if } X(p) = X(q) \wedge Y(p) < Y(q) \\ SE & \text{if } X(p) > X(q) \wedge Y(p) < Y(q) \end{cases}$$

- The development of cardinal directions between two moving points

When two points change their locations over time, the directional relation between them becomes time related, and may or may not change. First, we consider the cardinal directions at time instances. Let time denote the temporal data type representing time and MPoints denote the spatio-temporal data type that represents moving points. For $A, B \in$ MPoints, let $A(t)$ and $B(t)$ denote the snapshots of A and B at a time instance $t \in time$. If both A and B are defined at time t, then $A(t), B(t) \in Points$. The cardinal direction between A and B at t is therefore $dir(A(t),B(t)) \in CD$. For example, in Figure 3-29A, at time t_1 when A and B locate at $A(t_1)$ and $B(t_1)$, the cardinal direction between A and B at time instance t_1 is $dir(A(t_1), B(t_1))=NW$. At the time instance t_2 when A and B move to $A(t_2)$ and $B(t_2)$, the cardinal direction between them becomes $dir(A(t_2), B(t_2))=SE$. We propose our solution to determine what happened in between and to answer the question whether there exists a time instance t ($t_1 < t < t_2$) such that $dir(A(t), B(t))=W$ in the following sections. This scenario shows that within a common time interval, we may get different cardinal directions at different time instances. However, the change of time does not necessarily imply the change of

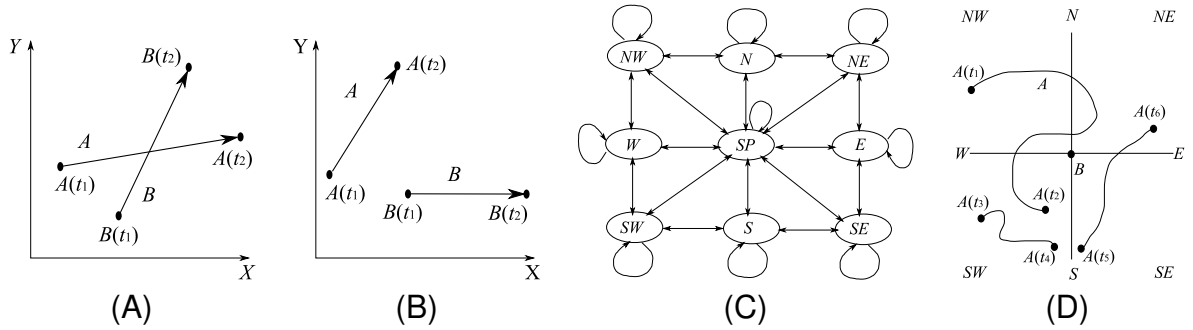


Figure 3-29. An example of two moving points with changed (A) and unchanged (B) cardinal directions over time. The state transition diagram of all cardinal directions (C), and an example of a cardinal direction development (D)

cardinal directions between two moving points. In Figure 3-29B two moving points A and B start at time t_1 from the locations $A(t_1)$ and $B(t_1)$ respectively. At time t_2, A reaches the location $A(t_2)$ and B reaches the location $B(t_2)$. One observation that we can obtain is that although the positions of A and B have changed, the cardinal direction between A and B does not change. In this case, A is always to the *northwest* of B between t_1 and t_2. In other words, the cardinal direction between two moving points holds for a certain period of time before it changes. Thus, we define a predicate *holds* that returns true if a cardinal direction holds during a time interval for two moving points A and B.

Definition 41. *Given two moving points $A, B \in MPoint$, a time interval $I = [t_b, t_e]$ with $t_b, t_e \in time$ and $t_b \leq t_e$, and the basic cardinal direction set CD. Assume both A and B are defined on I. Let holds be the predicate that returns true if $d \in CD$ holds over the time interval I for A and B. We define this function as*

$$holds(A, B, I, d) = \begin{cases} true & \text{if } \forall t \in I : dir(A(t), B(t)) = d \\ false & \text{otherwise} \end{cases}$$

For a given time interval I, we make the following observations: (i) if there exists a cardinal direction $d \in CD$ such that $holds(A, B, I, d) = true$, then we say A and B have a unique cardinal direction on the time interval I; (ii) if both A and B are defined on I, and the predicate *holds* returns *false* for all 9 basic cardinal directions in CD, then we say that A and B have a developing cardinal direction relationship over I; (iii) if neither A

nor B is defined on I, we say the cardinal direction between A and B is *not defined* on I. Therefore, we can only determine cardinal directions between A and B during intervals on which they are defined. For an interval where there is no unique basic cardinal direction that holds over the entire period, we split it into several sub-intervals such that on each sub-interval only a unique cardinal direction holds.

Further, if we regard different cardinal directions that hold over different sub-intervals as cardinal direction states, the development of the cardinal directions refers to a sequence of transitions between these states. For example, A moving from NW to W to SW of B is a development of cardinal directions between two moving points A and B. However, not all transitions are possible between any two states. Figure 3-29(C) shows all possible transitions between different states. For example, if the cardinal direction between two moving points A and B has been NW so far, then if time changes, the cardinal direction might stay the same as NW, or change to either N, W, or SP. It is not possible that A moves directly to the *south* (S) of B without crossing any other directions. This state transition diagram implies that only developments that involve valid transitions are possible between two moving points, e.g, A moves from NW to W to SW of B. Developments that involve invalid transitions like A moving from NW to S of B are not possible and thus not allowed. Let the predicate $isValidTrans : CD \times CD \rightarrow bool$ take two cardinal directions as input, and yield $true$ if the transition between them is valid. Then, for example, $isValidTrans(NW, W) = true$ while $isValidTrans(NW, S) = false$. Now we can define the development of the cardinal directions between two moving points A and B on any given time interval I on which A and B are both defined.

Definition 42. *Given two moving points $A, B \in MPoints$ and a time interval I on which both A and B are defined. Assume the ordering of any two intervals $I_1 = [t_{b1}, t_{e1}]$ and $I_2 = [t_{b2}, t_{e2}]$ is defined as $t_{e1} \leq t_{b2} \Leftrightarrow I_1 \leq I_2$. Let the symbol $\triangleright$ represent the transition*

from one cardinal direction state to another. Then the development of cardinal directions between A and B on interval I, denoted as $dev(A, B, I)$ can be defined as:

$$dev(A, B, I) = d_1 \triangleright d_2 \triangleright \ldots \triangleright d_n$$

if the following conditions hold:

(i) $n \in \mathbb{N}$ (ii) $\forall 1 \leq i \leq n : d_i \in CD$

(iii) $\forall 1 \leq i \leq n-1 : d_i \neq d_{i+1} \wedge isValidTrans(d_i, d_{i+1}) = true$

(iv) $\exists I_1, I_2, \ldots, I_n :$

(a) $\forall 1 \leq i \leq n : I_i$ is a time interval, $holds(A, B, I_i, d_i) = true$

(b) $\forall 1 \leq i \leq n-1 : I_i \leq I_{i+1}$,

(c) $\bigcup_{i=1}^{n} I_i = I$

In Definition 42, we split the given time interval into a sequence of non-overlapping sub-intervals. The development dev represents the transition of cardinal directions over these sub-intervals. Condition (iv)(a) ensures that a unique cardinal direction between A and B holds on each sub-interval, and condition (iv)(c) ensures that all sub-intervals together form a full decomposition of the given interval I. Further, according to condition (iii), only valid transitions are allowed between two cardinal directions that hold on adjacent sub-intervals. An example of such a development can be derived from Figure 3-29(D), where A moves from location a_1 to location a_2 and B does not move during the time interval $I = [t_1, t_2]$. The development of cardinal directions between A and B during I is therefore $dev(A, B, I) = NW \triangleright N \triangleright NE \triangleright N \triangleright NW \triangleright W \triangleright SW$. It describes that from time t_1 to time t_2, A starts in NW of B, crosses N and reaches NE of B, then it turns around and crosses N again, and returns to NW of B. Finally, A crosses W of B and ends up in the SW of B.

Now we are ready to define cardinal direction developments between two moving points during their entire life time. The idea is to first find out their common life time intervals, on which both A and B are defined. Then we apply the dev function to determine the development of cardinal directions between A and B during each

common life time interval. Finally, we compose the cardinal direction developments on different common life time intervals and define it as the development of cardinal directions between the two moving points A and B. We first find out the common life time intervals for two moving points.

Definition 43. *Given two moving points* $A, B \in MPoints$*, let* $LT_A = \langle I_1^A, I_2^A, \ldots, I_m^A \rangle$, $LT_B = \langle I_1^B, I_2^B, \ldots, I_n^B \rangle$ *be two life time interval sequences of* A *and* B *respectively such that* $I_i^A < I_{i+1}^A$ *and* $I_j^B < I_{j+1}^B$ *with* $1 \leq i \leq m, 1 \leq j \leq n$*. We can now define the common life time* CLT *for* A *and* B *as*

$$CLT(A, B) = \langle I_1, I_2, \ldots, I_l \rangle$$

if the following conditions hold:

$$\text{(i) } l < n + m \qquad \text{(ii) } \forall 1 \leq i \leq l : I_i \in \bigcup_{j=1}^{m} \bigcup_{k=1}^{n} (I_j^A \cap I_k^B) \qquad \text{(iii) } \forall 1 \leq i < l : I_i < I_{i+1}$$

Definition 43 defines the common life time of A and B as a list of intervals on which both A and B are defined. Directional relationships between A and B only exist during their common life time. At any time instance that is not within their common life time, the cardinal direction between A and B is not defined. For example, in Figure 3-29(D), since B is assumed to exist all the time, the common life time of A and B is $CLT(A, B) = \langle [t_1, t_2], [t_3, t_4], [t_5, t_6] \rangle$. During the time interval (t_2, t_3), the moving point A is not defined; thus it is not part of the common life time of A and B. Finally, the development of cardinal directions between A and B can be defined as

Definition 44. *Given two moving points* $A, B \in MPoints$ *and* $CLT(A, B) = \langle I_1, I_2, \ldots, I_l \rangle$*. Let the symbol* $\perp$ *represent the meaning of undefined direction. Then the development of cardinal directions between* A *and* B*, denoted as* $DEV(A, B)$ *can be defined as:*

$$DEV(A, B) = dev(A, B, I_1) \triangleright \perp \triangleright dev(A, B, I_2) \triangleright \perp \triangleright \ldots \triangleright dev(A, B, I_n)$$

Definition 44 generalizes the development of cardinal directions between two moving points from a given interval to their entire life time. The cardinal direction development between A and B in Figure 3-29(D) is therefore $DEV(A, B) = NW \triangleright N \triangleright NE \triangleright N \triangleright NW \triangleright W \triangleright SW \triangleright \perp \triangleright SW \triangleright \perp \triangleright SE \triangleright E \triangleright NE$.

CHAPTER 4
DISCRETE THREE-DIMENSIONAL DATA MODELING

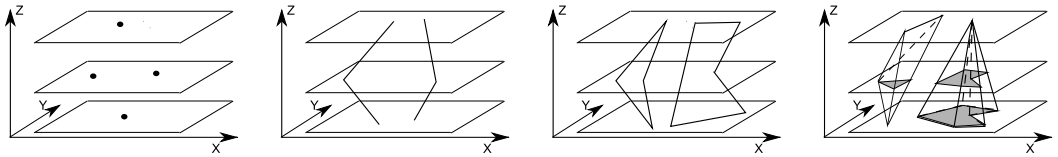
In Chapter 3, we have proposed a Spatial Algebra 3D at the Abstract Level (*SPAL3D-A*) to provide a formal specification for spatial objects in the 3D space and a set of important operations and predicates on them. The implementation of an abstract data model like *SPAL3D-A* requires the design of geometric data structures for the abstract data types, and the design of algorithms for operations and predicates. When designing the data representations and the algorithms in a database context for querying and analysis, more requirements need to be considered. First, the representations must be able to represent complex spatial objects like volumes with cavities, volumes with multiple components, and non-manifold surfaces (expressiveness). Second, the data structure designed for a spatial object must support efficient access of its components, so that point query and window query can be efficiently supported (efficiency). Third, the data structure designed for each single complex 3D spatial data type must be a single, general-purpose data structure that is universally applicable to a large range of 3D operations so that expensive data structure conversions can be avoided (adaptability). Fourth, when designing algorithms for operations, only external algorithms that do not require the entire object to be in main memory should be considered (scalability). Fifth, the algorithms should not assume special properties like convexity or monotonicity of a spatial object (generality).

In this chapter, we describe a discrete model, called the Spatial Algebra 3D at the Discrete Level (*SPAL3D-D*), that deals with the design of the finite representations for spatial objects and the design of algorithms for operations and predicates. Two data representations are proposed in this chapter. In Section 4.1, we first introduce a paradigm called slice representation as a general data representation method for all 3D spatial data types. We also present 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. In Section 4.2, we introduce a second tetrahedralization based compact data structure for *volume* objects in particular. The motivation for the second data representation is that volume data are available as a collection of tetrahedra due to the development of technologies like laser scanner. Although the tetrahedralizaion (TEN) based approach brings a lot of benefits like improving efficiency of spatial operations and reducing complexity in volume validations, a major issue is the expensive storage cost for maintaining the topological relations among tetrahedral mesh elements. We demonstrate that our TEN based data structure minimizes the storage for topological relations while maintaining the optimal topological relation access among mesh elements. Further, we present efficient algorithms for spatial queries like point query and window query and spatial operations like intersection operation based on our proposed data structure. Finally, we make a comparison between the two representations in Section 4.3. Further, as a detour, we develop algorithms for computing cardinal directions between two moving points in Section 4.4.

4.1 A Slice-based Representation of 3D Spatial Data

The definitions of the complex 3D spatial data types at the abstract level [89] 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



(a) *point3D* (b) *line3D* (c) *surface* (d) *volume*

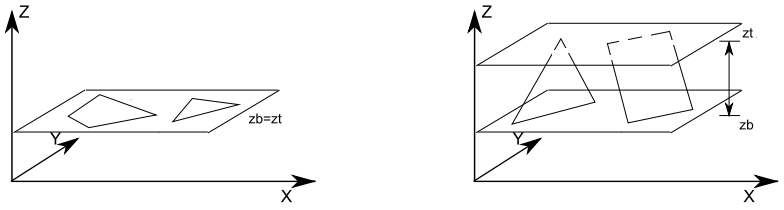
Figure 4-1. Slice representations of spatial objects

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.

4.1.1 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 into a sequence of none overlapping pieces, namely slices, by cutting the object on its n vertices with n' planes ($n \geq n'$), $P_1, P_2, \ldots, P_{n'}$, 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. A slice is a piece lying in between two adjacent cutting planes P_i and P_{i+1}. Therefore, each slice can be attached with a unique z-interval $I_z = \{(z_b, z_t) \mid z_b, z_t \in Real, z_b \leq z_t\}$, that identifies the location of that slice in the object



(a) an example of a thin slice

(b) an example of a thick slice

Figure 4-2. Two types of slices

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 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 4-2 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 $Point = \{(x, y, z) \mid x, y, z \in Real\}$. A value of the spatial data type *point3D* is a 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 $p_1 < p_2 \Leftrightarrow p_1.x < p_2.x \vee p_1.x = p_2.x \wedge p_1.y < p_2.y$. Then we can give the definition for th slice unit spoint:

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

The data structure is an array of an ordered 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 $Segment = \{(p, q) \mid p, q \in Point,\ p < q\}$. 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 $HSegment = \{(s, d) \mid s \in Segment,\ d \in \{left, right\}\}$. 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 with different dominating points. We store half segments and order

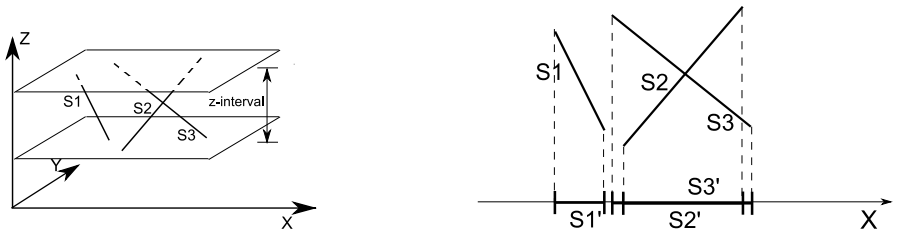
them with respect to their dominating points. For two half segments $h_1 = (s_1, d_1)$ and $h_2 = (s_2, d_2)$, let dp be the function which yields the dominating point of a half segment, and ndp be the function that returns the end point that is not a dominating point of the half segment. Then we define 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:

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

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

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) for *thin_sline3D* defines an empty z-interval and ensures that no segments in the slice have extent on z-direction. Condition (ii) 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 4-3 gives an example of a slice unit of a *line3D* object.

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



(a) segments in a slice (b) projections of segments on the x-axis

$$\langle (s1, \mathit{left}), (s1, \mathit{right}), (s3, \mathit{left}), (s2, \mathit{left}), (s2, \mathit{right}), (s3, \mathit{right}) \rangle$$

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

Figure 4-3. An example of a slice of a line3D object

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 z dimension, a simple polygon does not have holes, and is either a triangle or a trapezoid.

Lemma 13. *Let P denotes a simple polygon belonging to a thick slice s_i within a z-interval (z_i, z_{i+1}), where $z_i < z_{i+1}$, 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 exists more than three none co-linear 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:

$$tface = \{(p_1, p_2, p_3, p_4) \mid p_1, p_2, p_3, p_4 \in Point, p_1, p_2, p_3 \text{ and } p_4 \text{ are coplanar};$$
$$(\mathrm{i}) : p_1.z = p_4.z, p_2.z = p_3.z, p_1.z < p_2.z;$$
$$(\mathrm{ii}) : p_1 \leq p_4, p_2 \leq p_3\}$$

A $tface$ is represented with 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 left most point and the right most 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 left most point of a $tface$ and the right most 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 left most point or the right most point. A $tface$ therefore can be split into two half $tface$, left half $tface$ with the left extreme point to be the dominating extreme point and right half $tface$ with the right

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

$$\langle (F1, left), (F2, left), (F1, right), (F2, right), (F3, left), (F3, right) \rangle$$

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

Figure 4-4. An example of a thick slice of a surface object

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 *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*.

$$\begin{aligned} thick\_ssurface = \{(pos, \langle hf_1, \ldots, hf_n \rangle, it) \mid pos, n \in Integer, 0 \leq pos \leq n, n \geq 0; \\ \text{(i)} : \forall 1 \leq i < j \leq n : hf_i, hf_j \in Htface, hf_i < hf_j; \\ \text{(ii)} : \forall 1 \leq i \leq n : it \in l_z, hf_i.p_1.z = it.z_b, hf_i.p_2.z = it.z_t\} \end{aligned}$$

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 all *tfaces* in the thick slice to be ordered from left to right. Condition (ii) defines a none 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 4-4 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

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

The data structure for a thin slice of a surface object consists 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:

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

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 none-manifold, 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 t-faces 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 $< (*ff_1, *nc_1), \ldots, (*ff_m, *nc_m) >$ 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 $< *fc_1, \ldots, *fc_r >$ 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

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

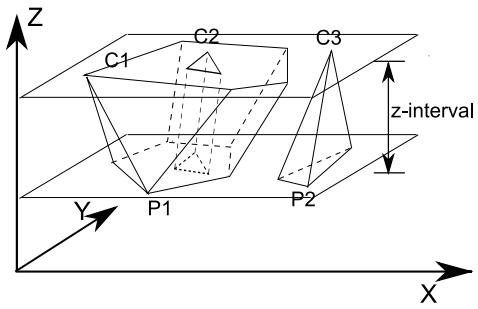


Figure 4-5. An example of a volume slice

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 4-5 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.

4.1.2 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:

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

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

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

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

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.

$\langle \alpha_1, \alpha_2, \alpha_3, \alpha_4 \rangle = \langle spoint, thin\_ssurface, thick\_ssurface, svolume \rangle$

$\langle \beta_1, \beta_2, \beta_3, \beta_4 \rangle = \langle point, HSegment, Htface, Htface \rangle$

$\langle \gamma_1, \gamma_2, \gamma_3 \rangle = \langle point3D, surface, volume \rangle$, $\langle \lambda_1, \lambda_2, \lambda_3 \rangle = \langle spoint, ssurface, svolume \rangle$

$\forall i \in \{1, 2, 3, 4, 5, 6\}$

get_first	: α_i	$\rightarrow \beta_i$	get_first_slice	: γ_i	$\rightarrow \lambda_i$
get_next	: α_i	$\rightarrow \beta_i$	get_next_slice	: γ_i	$\rightarrow \lambda_i$
end_of_array	: α_i	$\rightarrow bool$	end_of_seq	: γ_i	$\rightarrow bool$
$proj\_x$	: β_1	$\rightarrow (Real)$	is_empty	: γ_i	$\rightarrow bool$
	: β_i	$\rightarrow (Real, Real)(i \neq 1)$	$interval\_z$	: γ_i	$\rightarrow I\_z$

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 $n = 0, pos = 0, l_z = (0, 0)$. The other two operations, $insert\_point\_slice$ and $insert\_point$, insert a point slice and a point into data array respectively, and both increase pos and n by 1.

4.1.3 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.

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

For any of the operand combination, we introduce algorithms $sp3D\_intersect$ and $vp3D\_intersect$ separately.

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, and possible intersecting slice units are picked for testing. Then the problem becomes 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 *plane sweep* paradigm to find out all possible *tface-point* intersecting pairs within the two 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. Therefore,

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

Figure 4-6. The *sp3D_sweep* algorithm.

the sweeping algorithm *sp3D_sweep* for a thick surface slice *ss* and a point slice *sp* can be described in Figure 4-6.

The algorithm in Figure 4-6 computes intersection between a thick surface and a point slice. Figure 4-9 (a) 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

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

Figure 4-7. The $sp3D\_sweep'$ algorithm.

$sp3D\_sweep'$, which takes a thin slice ts and a point slice sp as operands. The algorithm is described in Figure 4-7.

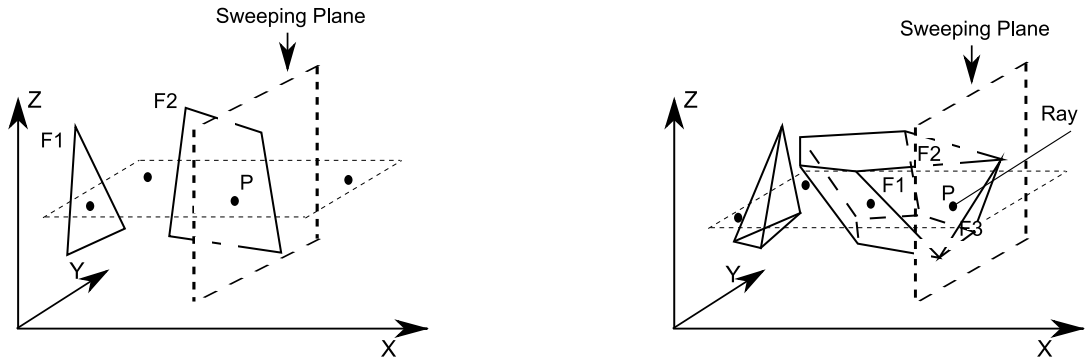
So far, we have handled both thin surface slices and thick surface slices, then we give the algorithm $sp3D\_intersect$ for a surface object s and a point3D object p in Figure 4-8.

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 s consists of v slices with either t $tfaces$ for each thick slice or t half segments for each thin slices. 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.

```
     method sp3D_intersect (s, p)
(1)  c ← new_point3D
(2)  if is_empty(p) and !is_empty(s)
(3)  then sp ← get_first_slice(p)
(4)     ss ← get_first_slice(s)
(5)     while !end_of_seq(sp)
(6)           and !end_of_seq(ss) do
(7)        if interval_z(sp)
(8)              > interval_z(ss)
(9)        then
(10)          ss ← get_next_slice(s)
(11)       else if interval_z(sp)
(12)             < interval_z(ss)
(13)       then
(14)          sp ← get_next_slice(p)
(15)       else if interval_z(sp)
                 == interval_z(ss)
(17)       then
(18)          sc ← sp3D_sweep(ss, sp)'
(19)          c ← insert_point_slice(c, sc)
(20)       else
(21)          sc ← sp3D_sweep(ss, sp)
(22)          c ← insert_point_slice(c, sc)
(23)       endif
(24)    endwhile
(25) endif
(26) return c
     end
```

Figure 4-8. The *sp3D_intersect* algorithm.

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 sequence 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 another word, 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



(A) a surface-point slice pair (B) a volume-point slice pair

Figure 4-9. Sweeping algorithms on slices

	method *vp3D_sweep* (sv, sp)		
(1)	$sc \leftarrow$ *new_point_slice*	(17)	$ray \leftarrow$ new ray shooting
(2)	$SL \leftarrow$ new empty hash	(18)	from p to $+\infty$ of the y-axis
(3)	$p \leftarrow$ *get_first(sp)*	(19)	$cnt \leftarrow 0$
(4)	$hf \leftarrow$ *get_first(ss)*	(20)	**for** $i = 0$ to the number of
(5)	**while** $!end\_of\_array(sp)$	(21)	elements in *sweep status list*
(6)	**and** $!end\_of\_array(sv)$ **do**	(22)	**if** $ray \cap SL[i]$ **then**
(7)	**if** $p \geq dep(hf)$ **then**	(23)	$cnt \leftarrow cnt + 1$
(8)	**if** $hf.d ==$left **then**	(24)	**endif**
(9)	insert hf in to the	(25)	**endfor**
(10)	hash array SL	(26)	**if** cnt is an odd number **then**
(11)	**else**	(27)	$sc \leftarrow insert\_point(p, sc)$
(12)	remove the twin of hf	(28)	**endif**
(13)	from the hash array SL	(29)	**endif**
(14)	**endif**	(30)	**endwhile**
(15)	$hf \leftarrow$ *get_next(sv)*	(31)	**return** sc
(16)	**else**		**end**

Figure 4-10. The *vp3D_sweep* algorithm.

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 4-10.

Above algorithm computes the intersection between a volume slice and a point slice. Figure 4-9B 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 $F1$,

```
     method vp3D_intersect (v, p)
(1)  c ← new_point3D
(2)  if !is_empty(p) and !is_empty(v)
(3)  then 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)
(8)        then sv ← get_next_slice(v)
(9)        else if interval_z(sp)
(10)              < interval_z(sv)
(11)       then sp ← get_next_slice(p)
(12)       else
(13)          sc ← vp3D_sweep(sv, sp)
(14)          c ← insert_point_slice(c, sc)
(15)       endif
(16)    endwhile
(17) endif
(18) return c
     end
```

Figure 4-11. The *vp3D_intersect* algorithm.

$F2$, 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 odd number of *tfaces*, P intersects the volume slice. Finally, we can give the *vp3D_intersect* algorithm in Figure 4-11.

According to the algorithm in Figure 4-11, the time complexity of *vp3D_intersect* algorithm is also the same as the *sp3D_intersect* intersection algorithms. Thus we ignore the analysis here.

4.2 A Compact TEN Based Data Structure for Complex Volumes

We have introduced the slice representation as the paradigm of designing data structures for spatial data types. We have given four data structures for all spatial data types 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. However, spatial objects like volumes are captured from various sources like laser scanner, sensors, and CAD softwares, each has its own format. As we described in Section 2.1, there exist two main representation approaches for volume objects, the boundary representation and the volume representation. The boundary representation approaches approximate the smooth surface of a volume with connected polygonal faces that encloses a subset of 3D space. Slice representation is one such approach.

One major disadvantage of the boundary representations is that they ignore the interior and is not suitable for volume based computations. Further, the conversion from volume representations to boundary representations like the slice representation is rather expensive. Thus, for the data type *volume*, we employ a decomposition strategy for its interior, called tetrahedralization, that decomposes a volume object into a collection of non-overlapping tetrahedra.

A tetrahedron is the simplest volume object in 3D space that can be uniquely determined by four vertices. As a result, operations on complex volumes can be converted to composing results from operations on tetrahedra, which reduces the complexity of the operations. The objective of this section is to present a compact data structure for representing and manipulating complex volume objects based on its tetrahedralization (TEN) embedded in the 3D Euclidean space. We first review some background information of the storage and query processing for tetrahedral meshes in Section 4.2.1. In Section 4.2.2, we propose a new data structure, that we call the connectivity encoded tetrahedral mesh (CETM), which provides a compact storage for complex volumes, i.e., volumes possibly with non-manifold components or cavities, and we present its support of efficient navigation operators for traversing elements in the mesh in Section 4.2.3. Finally, we evaluate the storage cost of our CETM algorithmically in Section 4.2.4.

4.2.1 Background

Most of the existing work for representing three-dimensional objects are boundary based representations. The boundary of a three-dimensional object is either discretized as polygonal meshes or triangle meshes. Data structures that are suitable for polygonal(triangle) meshes include the *Winged-Edge* [9], the *Half-Edge* [67], the *DCEL* [73], and the *Quad edge* [46]. These data structures share in common that they maintain the association among edges, bounding vertices and incident faces. Data structures that support more general polygonal(triangle) meshes, i.e., non-manifold

polygonal meshes and non-regularized polygonal meshes, have been proposed in *Facet-Edge* [30] and *Handle-Faces* [64], which are extensions of the *Quad-Edge* and the *Half-Edge* respectively.

All above data structures for polygonal(triangle) meshes store connectivity information among mesh elements, e.g. vertices, edges and faces, thus support efficient query and traversal operators. However, they consume large storage [56] when the mesh grows. Compact representations that are customized for triangle meshes are proposed. Examples are the edge based representation called Directed Edges [17], the *Star-Vertices* representation [56] that stores only the sorted list of references to neighbors for each vertex, and the Conner Table approach [54] that represents a manifold triangle mesh with a conner table and an opposite conner table. Although polygonal(triangle) meshes are more suitable for visualization of three-dimensional objects, they are not suitable for operations like ray-tracing, intersection, and volume computation. Therefore, volume based data structures are needed for handling these complex operations.

Tetrahedral meshes, also known as three-dimensional simplicial complexes, is another powerful tool for describing arbitrarily shaped three-dimensional objects, and is able to support efficient volume based queries and operations. Several data structures have been customized for tetrahedral meshes. Extensions of the Corner Table approach have been proposed independently in the Vertex Opposite Table (VOT) [12] approach and the *Compact Half Faces* (CHF) [60] approach to represent tetrahedral meshes. The VOT requires 8 references per tetrahedral, 4 for vertex references and 4 for opposite corners. Two corners are opposite if they belong to two adjacent tetrahedra and are not on the shared triangle face. References to the opposite corners are used to provide constant cost access to adjacent tetrahedra and their bounding cells. An improved version of the VOT approach called the *Sorted O Table* (SOT) is introduced in [49] which reduces the storage requirements to only 4 references and 9 service bits

per tetrahedron. The improvement of storage cost is achieved by eliminating the V table in the VOT entirely. The values in the removed V table can be inferred from the information stored in the O table and the service bits. The SOT approach provides compact storage and supports the efficient traversal operators. However, it does not support non-manifold tetrahedral meshes, thus are not suitable for complex volume objects. Other compression strategies exist for tetrahedral mesh [19, 47, 96] to reduce the storage cost. However, random access and traversal operators on the mesh are not supported in these approaches. A recent approach [] utilizes a spatial index Octree for storing tetrahedral meshes and allows local reconstruction of topological relations at run time. It combines the topological data structure with the spatial index so that it offers the flexibility of using the data structure for both spatial and topological queries. Although its experimental study shows the cost of reconstructing local topological data structure is amortized over multiple accesses to elements with in the same leaf node, the random access to any arbitrary topological relation is still costly. Therefore, it trades the topological relations access time for storage efficiency and flexibility. As a result, it does not support optimal incident relations retrieval even for manifold tetrahedral meshes, which is crucial to most visualization applications.

Dimension-independent data structures that stores n-dimensional complexes have been proposed in Indexed data structure with Adjacencies (IA) [75, 77], the *n-Generalized Maps*[62], the *Simplified Incident Graph* [27], and the *Non-Manifold Indexed with Adjacencies* (NMIA) [28]. All the data structures that are designed for n-dimensional complexes are also suitable for representing tetrahedral meshes. A good review about these data structures and the comparisons among them can be found in [55] and [29]. A problem with these approaches is that since they are the dimension independent approaches, the generalization to n-dimensional complexes introduces more overhead. Thus, they result in more storage cost [56].

There is very little work in the literature on techniques for performing spatial queries on tetrehedral meshes. Mainly there are two strategies for improving the query efficiency. The first is to extend multidimensional spatial indexes like R-Tree and Octree to speed up the search on mesh elements. An excellent survey on this topic is available in [43]. R-Tree based indexes approximates objects by their minimum bounding boxes (MBBs) and index them in a hierarchical tree structure [50]. Variants like the Hilbert-packed R-Tree [57], the Priority R-Tree [4] and others [10] attempt to improve the organization of datasets in R-Tree for better performance. However, tetrahedral meshes are a challenging application for R-Trees because the tetrahedra in a mesh are mostly adjacent or incident, which results in a significant amount of overlapping bounding boxes. This has a large impact on the efficiency of the search queries on such indexes. Non-overlapping R-Tree variant, the R+-Tree [90], solves this problem by generating disjoint MBBs and replicating the objects that cross MBB boundaries. However, this approach significantly increases storage requirements, which also affects the query performance.

Another index approach is to extend Octrees or kd-trees for tetrahedral meshes. They are based on the recursive subdivision of an initial cubic domain through planes passing through the center of the block into eight or into two blocks for Octrees and kd-tress respectively [86]. Based on different subdivision criteria and different content in the leaf nodes, the Octrees and kd-trees can be extended to different spatial indexes for tetraheral meshes. A good review and comparison among different Octree and kd-tree based spatial indexes for tetrahedral meshes can be found in [36]. The point queries and window queries gain their efficiency purely based on the spatial locality information of the tetrahedral mesh elements stored in the indexes. However, we observe that the topological relations among tetrahedral mesh elements are also helpful for these spatial queries, and can even reduce the storage cost of these index structures.

In contrast with the tree index strategy that improves the query efficiency by indexing the spatial locality information of mesh elements, the other approach is to directly make use of the topological relations among meshes elements for efficient query processing. The Directed Local Search (*DLS*) approach in [79] applies the Hilbert curve to obtain an initial approximation solution, which is refined through local search algorithms that collect results by searching through adjacent neighbors of the approximated solution. This approach avoids the complexities involved in trying to capture the geometry of the mesh, by utilizing the topological relations among mesh elements. Although it is a light-weighted approach that does not require much storage overhead, it suffers from the following problems: first, when the mesh represents a skewed volume, a concave volume, or a volume with large cavities, DLS fails to find correct results; second, when the query results are empty which can be immediately detected by tree index approaches, DLS will yield a lot of unnecessary checks.

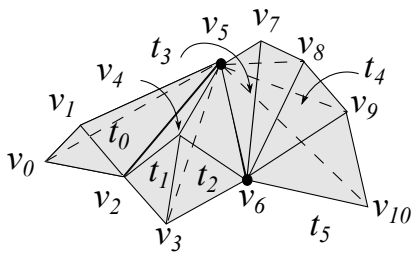
4.2.2 The Connectivity Encoded Tetrahedral Mesh (CETM)

A variety of data structures have been proposed for storing a tetrahedral mesh and its connectivity information which simplifies and accelerates common queries and traversal operators needed to support applications. Such queries and operations involving connectivity information are fundamental for many tasks which require local navigation on the mesh elements. Examples are visibility operations, mesh analysis and repair, ray tracing and boundary reconstruction. However, the problem rises when the the number of tetrahedra grows fast in a mesh. In some applications, a mesh can easily contain millions of tetrahedra, which results in an even larger storage for connectivity information. Therefore, a storage efficient data structure is desired for applications with meshes. Tetrahedral meshes compression schemes have been proposed for this purpose. But unfortunately, the compressed formats are not suitable for efficient traversal operators. Thus, a data structure that provides compact storage for tetrahedral

mesh while preserving the support for efficient random access operators and traversal operators are highly demanded.

In this paper, we propose our solution aiming at the following goals: 1) be able to describe most complex volume objects (expressiveness); 2) to reduce storage cost (compactness); 3) to support efficient retrieval of incidence and adjacency relations among elements in the mesh (efficiency). Our connectivity encoded tetrahedral mesh (CETM) stores a list of all vertices in the mesh, and for each tetrahedra, we store 4 references to the vertices that are its conner points. In addition to that, we introduce an optimal encoding strategy for the connectivity information between a tetrahedra pair. To reduce storage, we only store a minimal set of connectivity encodings, called spanning connectivity encodings, that are sufficient for efficient traversal operators. Algorithms for identifying such a set, as well as the algorithms for the traversal operators based on the connectivity encodings are also presented.

In this section, we describe in details our approach for representing general tetrahedral meshes that are possibly with non-manifold elements, the connectivity encoded tetrahedral mesh (CETM). In contrast to other approaches that maintain connectivity information among different mesh elements like vertices, edges, triangles and tetrahedra, which usually results in redundant storage, we explore all possible connectivity interactions between tetrahedra, encode them with four bits, and only store these connectivity encodings between tetrahedra. We further optimize the storage of the connectivity encodings by identifying the spanning connectivity encodings, which are a minimal set of connectivity encodings that are necessary for deriving all possible connectivity information in the tetrahedral mesh, and we store only this set of encodings. We prove that the connectivity information that is stored is also enough for deriving the relations among other mesh elements like vertex-edge, edge-triangle, etc.



V : $[v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}]$

TV : $[0, 1, 2, 5, 2, 3, 4, 5, 3, 4, 5, 6, 5, 6, 7, 8, 5, 6, 8, 9, 5, 6, 9, 10]$

$\uparrow$ t_0 $\uparrow$ t_1 $\uparrow$ t_2 $\uparrow$ t_3 $\uparrow$ t_4 $\uparrow$ t_5

Figure 4-12. An example of a tetrahedral mesh and its representation (assume $v_i < v_j$ for $0 \leq i < j \leq 10$).

4.2.2.1 Storing a Tetrahedral Mesh

We apply a data structure that stores all vetices and the Tetrahedron-Vertex relations, i.e., the relation among a tetrahedron and its four vertices, in a tetrahedral mesh. Two arrays are used for this purpose: 1) the vertices array V that stores all vertices in the mesh in terms of their coordinates in $\mathbb{R}^3$; 2) the tetrahedron-vertex array TV that stores four conner points for each tetrahedra in the mesh in terms of four references to the corresponding vertices in the V array. In the tetrahedron-vertex array TV, we order the four conner points of a tetrahedron in (x,y,z)-lexicographic order, so that every tetrahedron has a unique representation. For any two vertex points $p_1, p_2 \in Point$, we define a (x,y,z)-lexicographic order: $p_1 < p_2$, if and only if $p_1.x < p_2.x \vee (p_1.x = p_2.x \wedge p_1.y < p_2.y) \vee (p_1.x = p_2.x \wedge p_1.y = p_2.y \wedge p_1.z < p_2.z)$. In addition, we also sort the vertices in the V array in the same order so that $V[i] < V[j]$, where $0 \leq i < j < n_v$ and n_v is the total number of vertices in the mesh. Figure 4-12 gives an example of a tetrahedral mesh and its corresponding representation in V array and TV array. In this example, the V array stores all vertices from v_0 to v_{10} in order, while in the TV array, tetrahedra t_0 to t_5 are stored in order, each with four references linking to its four vertices in the V array.

This representation allows random access to ith tetrahedron in the mesh in constant time. It also allows the access to a tetrahedron's vertices, edges, and triangle faces in constant time. Below we list the access operators and their implementations on this representation:

$$
\begin{aligned}
getPoint(i) &:= V[i];\\
getTetra(t) &:= \langle TV[4t],\, TV[4t+1],\, TV[4t+2],\, TV[4t+3]\rangle\,;\\
getV(t,i) &:= TV[4t+i];\\
getE(t,i,j) &:= \langle getV(t,i), getV(t,j)\rangle\,;\\
getF(t,i) &:= \langle getV(t,(i+1)\%4), getV(t,(i+2)\%4), getV(i,(i+3)\%4)\rangle;
\end{aligned}
$$

The implementations of the operators are trivial. The $getPoint(i)$ function returns the ith point in the vertex V array. The $getTetra(t)$ returns the tth tetrahedra in the mesh in terms of a list of the references to its four conner points. The $getV(t,i)$, $getE(t,u,j)$ and $getF(t,i)$ operators returns the ith vertex in the tth tetrahedron, the edge with ith and jth vertices as end points in the tth tetrahedron, and the face that is in front of the ith vertex, that is, the face that does not incident on the ith vertex, in the tth tetrahedron respectively. For example, in the tetrahedral mesh in Figure 4-12, we can access the triangle face $\langle v_4, v_5, v_6\rangle$ of tetrahedron t_2 by calling $getF(2,0)$ which returns a list of references $\langle 4,5,6\rangle$. Further, if we want to get the actual vertices, we can use the $getPoint$ operator so that $\langle getPoint(4), getPoint(5), getPoint(6)\rangle = \langle v_4, v_5, v_6\rangle$.

4.2.2.2 Encoding Connectivity Information in a Tetrahedral Mesh

The data structure for the vertices (the V array) and the tetrahedron-vertex relations (the TV array) enables the storage of a tetrahedral mesh. However, the access of the relations among the mesh elements such as locating all tetrahedra that are incident at a given vertex and locating all triangle faces that are sharing a given edge is not efficient without additional connectivity information being stored. Thus, a lot of approaches explicitly store connectivity information like vertex-edge relations and vertex-face relations, which increases storage dramatically. Therefore, we propose our approach

that encodes all possible connectivity that can happen between two tetrahedron and stores only the encoded connectivity at the tetrahedron level. We prove later that our encoding strategy leads to compact storage by comparing with other approaches in Section 4.2.4.2, and we show that it still maintains the efficiency for traversal operators in Section 4.2.3.

In this section, we introduce the connectivity encodings that describes how two terahedra are connected, at a vertex, an edge, or a triangle, and further which vertex, edge, or triangle. We first define the connectivity encoding function between two tetrahedra in a tetrahedral mesh in Definition 45

Definition 45. *Let t_i, t_j be the ith and jth tetrahedra in a tetrahedral mesh stored in the TV array, and let $\iota(i,j,k)$ denote the function that checks whether t_i and t_j share the kth vertex point in the V array, then we have:*

$$\iota(i,j,k) = \begin{cases} 1 & \text{if } \exists 0 \leq \alpha, \beta < 4 : getV(i,\alpha) = getV(j,\beta) = k \\ 0 & \text{otherwise} \end{cases}$$

Now, let the function $CE(i,j)$ denote the function that generates the connectivity encoding between t_i and t_j with respect to t_i, then we have:

$$CE(i,j) := \left(\iota(i,j,getV(i,0)) \;\; \iota(i,j,getV(i,1)) \;\; \iota(i,j,getV(i,2)) \;\; \iota(i,j,getV(i,3)) \right)$$

Definition 45 provides an encoding strategy that encodes the connectivity between two tetrahedron using only four bits. Please note that the connectivity encoding function $CE(i,j)$ is an asymmetric function that takes t_i as the reference tetrahedron and encodes the connectivity interaction with the target tetrahedron t_j with respect to the vertices of the reference tetrahedron t_i. As a result, $CE(i,j)$ and $CE(j,i)$ may be different. The type of connectivities, i.e. vertex sharing, edge sharing, or triangle face sharing, can be determined by the number of 1s in the encoding, and the position of 1s indicates the index of the shared vertex in a tetrahedron. Figure 4-13 shows a total of 14 possible neighborhood connectivities and their corresponding

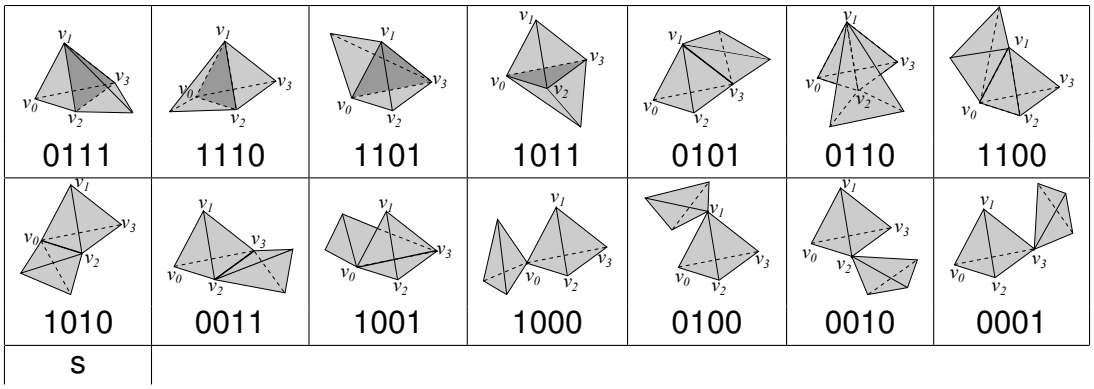


Figure 4-13. A total of 14 possible connectivity interactions and their corresponding encodings for a tetrahedron $\langle v_0, v_1, v_2, v_3\rangle (v_0 < v_1 < v_2 < v_3)$ with some other tetrahedron.

encodings that can happen for the reference tetrahedron $t = \langle v_0, v_1, v_2, v_3\rangle$ in a tetrahedral mesh (the disjoint case 0000 and the equal case 1111 are omitted). This enables us to describe the connectivity between two given tetrahedron with a pair of connectivity encodings. For example, in Figure 4-12, the tetrahedra t_1 and t_2 share a triangle face $\langle v_3, v_4, v_5\rangle$, thus the corresponding connectivity encoding for them is $(CE(1,2), CE(2,1))$=((0111),(1110)). Moreover, if we are given two connectivity encodings $CE(1,0)$=(1001) and $CE(1,2)$=(0111), then we can derive the relation between t_0 and t_2 that they share the fourth vertex of t_1 (the vertex v_5) without finding out the actual connectivity between t_0 and t_2. This is a very useful feature that is extensively used in implementing operations (Section 4.2.3).

As a result, in addition to the two arrays, i.e., the V array and the TV array, that we have introduced in Section 4.2.2.1 for storing a tetrahedral mesh, we need an extra data structure that stores for each tetrahedron, a list of references pointing to the tetrahedra that it is connected together with the corresponding connectivity encodings. With this information stored, queries like "find all tetrahedra connecting with tetrahedron t" and "find all tetrahedra that connects with the vertex v of a tetrahedron t" can be answered efficiently. However, in a large tetrahedral mesh, each tetrahedron can connect with a large number of neighbors, especially when non-manifolds are allowed in the mesh.

Therefore, storing all connectivity encodings can be rather storage costly and is unnecessary. For example, in Figure 4-12, each tetrahedron has 5 connected neighbors. Thus if we store all connectivity encodings, a total of 30 connectivity encodings will be stored for the entire tetrahedral mesh. In Section 4.2.2.3, we propose an optimization strategy aiming at finding a minimal number of encodings that needs to be stored, while not affecting the efficiency of traversal operators.

4.2.2.3 Optimizing the Number of Stored Connectivity Encodings

The encoding introduced in the previous section describes how two tetrahedra are connected by using only four bits, thus is a compact representation for connectivity information in a tetrahedral mesh. The problem is that we can not afford the storage to store the connectivity encoding between any tetrahedron pair in the tetrahedral mesh. More importantly, it is not necessary.

The connectivity interactions among different tetrahedron pairs are not mutually exclusive in the sense that some can be derived from others. For example, let $c_{ij} = CE(i, j)$ denote the connectivity encoding between the ith and jth tetrahedra, then all 30 connectivity encodings in Figure 4-12 are listed in Figure 4-14A. However, the encoding $c_{45} = 1101$ and $c_{43} = 1110$ indicate that t_5 shares triangle $\langle v_5, v_6, v_9 \rangle$ with t_4, and t_3 shares triangle $\langle v_5, v_6, v_8 \rangle$ with t_4. It is not difficult to conclude that t_5 connects t_3 on edge $\langle v_5, v_6 \rangle$, which leads to the values of two other encodings c_{53} and c_{35}. Therefore, we call the connectivity encodings like c_{53} and c_{35} the derivable encodings from c_{45} and c_{43}. We formally define such encodings in Definition 46.

Definition 46. *Let* $CES = \bigcup_{0 \le i,j < n, i \ne j, c_{ij} \ne 0000} (i, j, c_{ij})$ *denote a set of triplets in a tetrahedral mesh with* n *tetrahedra such that each triplet describes a tetrahedra pair and their connectivity encoding. Let the set* C *denote a subset of* CES *(*$C \subset CES$*). We say that the connectivity encoding* c_{ij} *for the* i*th and* j*th tetrahedra is derivable from* C*, denoted as* $der(i, j, C)$*, if the following condition is satisfied.*

c_{ij}	t_0	t_1	t_2	t_3	t_4	t_5
t_0	-	0011	0001	0001	0001	0001
t_1	1001	-	0111	0001	0001	0001
t_2	0010	1110	-	0011	0011	0011
t_3	1000	1000	1100	-	1101	1100
t_4	1000	1000	1100	1110	-	1101
t_5	1000	1000	1100	1100	1110	-

(A)

c_{ij}	t_0	t_1	t_2	t_3	t_4	t_5
t_0	-	0011	-	-	-	-
t_1	1001	-	0111	-	-	-
t_2	-	1110	-	0011	-	-
t_3	-	-	1100	-	1101	-
t_4	-	-	-	1110	-	1101
t_5	-	-	-	-	1110	-

(B)

Figure 4-14. The complete set of connectivity encodings between ith tetrahedron (the row headers) and the jth tetrahedron (the column headers) in (A) and the spanning connectivity encodings in (B).

$$der(i,j,C) = \begin{cases} true & \text{if } c_{ij} \in C \vee (\exists 0 \leq k < n, k \neq i, k \neq j : ((c_{ik} \in C) \wedge \\ & (der(j,k,C - \{(i,k,c_{ik}),(k,i,c_{ki})\}) = true) \wedge \\ & (c_{ik} \& c_{ij} = c_{ij}) \wedge (c_{jk} \& c_{ji} = c_{ji})) \\ false & \text{otherwise} \end{cases}$$

Definition 46 describes a correlation between connectivity encodings. Therefore, our goal is to find out a minimal set of connectivity encodings C_{min} so that all connectivity encodings can be derived from the encodings in C_{min}. We call such set the spanning connectivity encoding set. The formal definition is given in Definition 47.

Definition 47. *Let* $CES = \bigcup^{0 \leq i,j < n, i \neq j, c_{ij} \neq 0000} (i,j,c_{ij})$ *denote a set of triplets in a tetrahedral mesh with* n *tetrahedra such that each triplet describes a tetrahedra pair and their connectivity encoding. Let the set* C_{min} *denote a subset of* CES ($C_{min} \subset CES$). *We call* C_{min} *the spanning connectivity encodings set if* C_{min} *satisfies the following three conditions: 1) for any triplet* (i,j,c_{ij}) *in* CES, c_{ij} *is derivable from the set* C_{min}, *i.e.,* $der(i,j,C_{min}) = true$; *2) for any triplet* (k,l,c_{kl}) *in* C_{min}, c_{kl} *is not derivable from the rest of* C_{min}, *i.e.,* $der(k,l,C_{min} - \{(k,l,c_{kl})\}) = false$; *3) if the triplet* (i,j,c_{ij}) *is in* C_{min}, *then the triplet* (j,i,c_{ji}) *is also in* C_{min}.

According to Definition 47, the spanning connectivity encodings set can derive the complete set of connectivity encodings that can exist in the tetrahedral mesh

```
algorithm FindSpanConEncoding (CES)
(1) spanlist ← new n × n boolean array
(2) for i = 0 to n
(3)    for j = 0 to n
(4)       spanlist[i][j] = true
(5)    endfor
(6) endfor
(7) for i = 0 to n
(8)    for j = i + 1 to n
(9)       if isDerivable(i, j, CES, spanlist)
(10)         spanlist[i][j] = false
(11)         spanlist[j][i] = false
(12)      endif
(13)   endfor
(14) endfor
(15) return spanlist
```

```
algorithm isDerivable (i, j, CES, spanlist)
(1) spanlist[i][j] = false, spanlist[j][i] = false
(2) result = false
(3) for k = 0 to n
(4)    if CES[i][k]&CES[i][j] = CES[i][j]
(5)    and CES[j][k]&CES[j][i] = CES[j][i]
(6)       and spanlist[i][k] then
(7)       if spanlist[j][k] then
(8)          result = true
(9)       else
(10)         spanlist[i][k] = false
(11)         spanlist[k][i] = false
(12)         if isDerivable(j, k, CES, spanlist)
(13)         result = true
(14)         endif
(15)         spanlist[i][k] = true
(16)         spanlist[k][i] = true
(17)      endif
(18)      if result then return true
(19)      endif
(20)   endif
(21) endfor
(22) spanlist[i][j] = true, spanlist[j][i] = true
(23) return result
```

Figure 4-15. The *FindSpanConEncoding* algorithm and the *isDerivable* algorithms.

(Condition 1), and at the same time, it is guaranteed to contain a minimal number of connectivity encodings since the removal of any encoding from the set causes the removed encoding to be not derivable from the remaining of the set (Condition 2). Further, since the connectivity encoding is asymmetric, the third condition ensures that the connectivity information is stored for both tetrahedra. Now the problem is how such a spanning connectivity encodings set can be found given a complete set of connectivity encodings of a tetrahedral mesh from the algorithmic perspective. We propose an algorithm, *FindSpanConEncoding*, in Figure 4-15 that identifies a collection of connectivity encodings that satisfies the requirement of a spanning connectivity encodings set. The algorithm takes a two dimensional $n \times n$ array CES as input, in which each element $CES[i][j]$ stores a 4 bit connectivity encoding for the ith tetrahedron and

V : $[v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}]$
TV : $[0, 1, 2, 5, 2, 3, 4, 5, 3, 4, 5, 6, 5, 6, 7, 8, 5, 6, 8, 9, 5, 6, 9, 10]$
$\uparrow$ t_0 $\uparrow$ t_1 $\uparrow$ t_2 $\uparrow$ t_3 $\uparrow$ t_4 $\uparrow$ t_5
TE : $[0, 1, 3, 5, 7, 9]$
E : $[0011, 1001, 0111, 1110, 0011, 1100, 1101, 1110, 1101, 1110]$
ET : $[1, 0, 2, 1, 3, 2, 4, 3, 5, 4]$

Figure 4-16. The data structure for the tetrahedral mesh in Figure 4-12 (assume $v_i < v_j$ for $0 \leq i < j \leq 10$).

jth tetrahedron. An additional list $spanlist$ is used to mark the current set of candidate spanning connectivity encodings. It indicates that $CES[i][j]$ is a candidate connectivity encoding if $spanlist[i][j]$ is a $true$ value. We initialize the candidate spanning connectivity encoding set to be the complete set, and we check every element in the set and remove it from the set by setting its corresponding flag in $spanlist$ to be $false$ if this element is derivable from the rest of the set. This strategy works because the removal of an element from the current set $spanlist$ does not affect the removed ones. For example, assume the current candidate spanning connectivity encoding set is S, and assume we have two encodings e_1, e_2 such that $e_1 \in S$ and $e_2 \notin S$. Thus, e_2 is derivable from S. Then if e_1 is derivable from $S - \{e_1\}$, e_2 is also derivable from $S - \{e_1\}$. We also implement the $isDerivable$ method to check whether a given encoding $CES[i][j]$ is derivable from a given encoding set identified by $spanlist$.

As a result, the algorithm identifies a collection of connectivity encodings that satisfies the requirements of a spanning connectivity encoding set in Definition 47. In our example in Figure 4-12, a total of 30 connectivity encodings exists (Figure 4-14A). After running the algorithm, a total of 10 spanning connectivity encodings are identified in Figure 4-14B. This is the minimal number of connectivities that need to be stored since no more encodings can be removed from this set, otherwise once removed, it can not be restored from the rest of the set. The algorithm that we introduced can be further optimized in terms of efficiency by applying some greedy rules like only keeping one

encoding out of a group of encodings that are derivable from each other. One of our future work is to develop an optimized algorithm for finding out the spanning connectivity encoding set. However, in this paper, this is beyond our scope.

In order to hold such spanning connectivity encodings for a tetrahedral mesh, we introduce extra data structures. In addition to the two data arrays V and TV that holds all vertices in the mesh and all tetrahedra in the mesh respectively, we use three more arrays, the TE array, the E array, and the ET array. The E array stores the 4 bit spanning connectivity encodings for each tetrahedron, and the spanning connectivity encodings (c_{i*}) that belong to the same tetrahedron (the ith tetrahedron) are stored consecutively. The TE array stores for each tetrahedra the position of its first spanning connectivity encoding in the E array. Finally, for the encoding c_{ij} between the ith and jth tetrahedra, if we call the ith tetrahedron the reference tetrahedron and the jth tetrahedron the target tetrahedron, then the ET array stores for each encoding the index of its target tetrahedron. Based on these new arrays, for any ith tetrahedron, we can reach the spanning connectivity encodings c_{i*} in constant time. The complete data structure for the tetrahedral mesh described in Figure 4-12 is presented in Figure 4-16. In this example, the connectivity encodings for t_2 are stored in $E[TE[2]]$ and $E[TE[2]+1]$. We can further retrieve the corresponding target tetrahedron index from $ET[TE[2]]$ (t_1) and $ET[TE[2]+1]$ (t_5). Below we list the access operators for the connectivity information stored and their implementations based on this representation:

$$countEncoding(t) := TE[t+1] - TE[t];$$

$$getEncoding(i,t) := \begin{cases} E[TE[t]+i] & \text{if } i < countEncoding(t) \\ 0000 & \text{otherwise} \end{cases}$$

$$getETetra(i,t) := \begin{cases} ET[TE[t]+i] & \text{if } i < countEncoding(t) \\ -1 & \text{otherwise} \end{cases}$$

The operator $countEncoding(t)$ returns the number of encodings stored for the tth tetrahedron. The operator $getEncoding$ retrieves the ith stored encoding for the tth

tetrahedron, and the operator $getETetra$ retrieves the corresponding target tetrahedron for the ith encoding of the t the tetrahedron.

4.2.3 Operators for Retrieving Topological Relations

One of the important tasks that is usually performed on a tetrahedral mesh is to traverse the mesh and to access the various elements (vetices, faces, and tetrahedra). The efficiency of these tasks are crucial to some applications. For example, the 3D ray tracing applications requires the efficient retrieval of the adjacent tetrahedra, the volume simplification algorithms which are based on the edge collapse strategy require the efficient traversal among adjacent edges in a tetrahedral mesh, and some multi-resolution 3D visualization tools requires the efficient identification of adjacent vertices so that they can be merged for a lower resolution display. Therefore, it is important that the data structures that store tetrahedral meshes supports the efficient access of these relations among different elements. In Section 4.2.3.1, we first present the definitions of a complete set of topological relations among the tetrahedral elements. We then show in Section 4.2.3.2 how the topological relations among the elements can be retried in optimal time from our compact CETM data structure.

4.2.3.1 Topological relations among tetrahedral mesh elements

A tetrahedral mesh contains four types of elements, vertices, edges, triangle faces, and tetrahedra. The topological relationships refer to the relationships like adjacency, boundary, and co-boundary among the same or different types of elements. A rather complete summary of all the topological relations among mesh elements is presented in [29, 37]. They describe the topological relations in a more general mesh called simplicial complexes which can be used to describe objects in any dimensional space. In this paper, we are only interested in the 3D tetrahedral mesh, thus we review the relevant concepts in a three-dimensional space.

In the tetrahedral mesh, a vertex, an edge, a triangle face, and a tetrahedron are 0-dimensional, 1-dimensional, 2-dimensional and 3-dimensional element respectively.

Let p, q denote the dimensionality of an element in a tetrahedral mesh ($0 \leq p, q \leq 3$) and let s denote a p-dimensional element in the tetrahedral mesh T. Then the topological relation function $R_{pq}(s)$ is defined as a retrieval function that returns the q-dimensional elements of the tetrahedral mesh T that are not disjoint from the p-dimensional element s. In particular:

- For $p < q$, $R_{pq}(s)$ retrieves all q-dimensional elements that share the p-dimensional element s, e.g., $R_{03}(s)$ retrieves all tetrahedra that share the vertex s.
- For $p > q$, $R_{pq}(s)$ retrieves all q-dimensional elements on the boundary of the q-dimensional element s, e.g., $R_{30}(s)$ retrieves all vertex points of the tetrahedron s.
- For $p > 0$, $R_{pp}(s)$ retrieves all p-dimensional elements in the mesh that shares a $p-1$-dimensional element with the q-dimensional element s, e.g., $R_{33}(s)$ retrieves all tetrahedra that share a triangle face with the tetrahedron s.
- $R_{00}(v)$, where v is a vertex, retrieves the set of vertices w such that $\{v, w\}$ is an edge element in the tetrahedral mesh.

The relation R_{pq} is called a boundary relation if $p > q$, a *co-boundary relation* if $p < q$ and an adjacency relation if $p = q$. Boundary and co-boundary relations together are called incident relations.

Since these topological relations are used extensively in a large amount of applications like visualization and shape reconstruction, it is very important for the data structures holding a tetrahedral mesh to support these topological relation functions efficiently. As we discussed in Section 4.2.1, several compact data structures have been proposed to reduce storage for tetrahedral meshes. However, they either do not support efficient topological relation functions [19, 47, 96] or only support the efficient operations for manifold tetrahedral meshes [49, 54]. Thus, in this section, we show how efficient topological relation functions can be implemented based on our connectivity encoded tetrahedral mesh (CETM).

4.2.3.2 Retrieving topological relations based on our CETM

We have given the notions for the topological relations in Section 4.2.3.1, we now present the implementations of these functions and we demonstrate the efficiency of the implementations based on our proposed CETM data structure by analyzing their time complexity.

First of all, the implementations for the boundary relations R_{pq} where $p > q$ based on our CETM are trivial. We have shown some basic operators in Section 4.2.2.1 that can be used to retrieve the boundary relations. Therefore, in this section, we focus more on the retrieval of co-boundary relations and adjacency relations.

The co-boundary relations retrieval functions are R_{01}, R_{02}, R_{03}, R_{12}, R_{13}, and R_{23}. These retrieval functions are also called the star operations. In principle, we can categorize these star operations in to three categories, the vertex star operations, the edge star operations, and the face star operations. The vertex star operations, denoted as R_{0*}, are the ones that retrieve edges, faces or tetrahedra sharing at a given vertex point. The edge star operations, denoted as R_{1*}, are the ones that retrieve faces and tetrahedra sharing at a given edge. The face star operations, denoted as R_{2*}, are the one that retrieves the tetrahedra sharing at a given face. We now first describe the implementation details for the vertex star operations.

The implementations of the vertex star operations R_{01} and R_{02} are trivial if the R_{03} operation is supported. One just needs to use the R_{03} operation to retrieve the tetrahedra sharing at a given vertex point, and for each tetrahedron retrieved, edges with the given vertex as an end point and faces with the given vertex point as one of its three conner points can be retrieved in constant time. So we only need to propose an efficient implementation strategy for the R_{03} operation. We first define a function that will be used in our algorithm. Let the function $getPos(t, v)$ denote a function that takes tth tetrahedra and a vertex point v as input, and returns the position of the vertex in the tetrahedra t. For example, in our example (Figure 4-12), $getPos(1, 3)$ retrieves the position of the

vertex v_3 in tetrahedron t_1, and returns 1 as result meaning that the position of v_3 in t_1 is 1. Now, we give the algorithm for retrieving the R_{03} relations in Figure 4-17. The *R03* algorithm takes the ith vertex of the tth tetrahedron as the input vertex and stores the resulting tetrahedra in the list $tlist$. In addition, a boolean $tflag$ array is used to keep track of the inserted tetrahdera and is initialized with 0 values. The algorithm first checks whether the current tetrahedra is in the list. If it is then stop searching (line 1-3), otherwise, the current tetrahedron is appended to the result list $tlist$ since the current tth tetrahedron always contains its ith vertex (line 4). Next, we check the connectivity encodings that are explicitly stored for the current tth tetrahedron. For those encodings that involve the ith vertex, we retrieve the corresponding target tetrahedra, and for each target tetrahedron, we recursively look for the tetrahedra that share the same vertex with them (line 8-13). Eventually, all tetrahedra that share at the given vertex will be stored in the list $tlist$. This algorithm works because the connectivity information stored are able to derive all other connectivity information that are not explicitly stored (Section 4.2.2.3). This algorithm has a $O(m)$ time complexity where m is the number of tetrahedra sharing at a given vertex.

Similarly, the implementation of the edge star operation R_{12} relies on the algorithm for the edge star operation R_{13}. Thus we only need to design the algorithm for retrieving R_{13} relations for a given edge. We present the algorithm *R13* for the R_{13} relation in Figure 4-17. The input of this algorithm is the same as the algorithm *R03* except that, instead of given the ith vertex of the tth tetrahedron, it gives an edge $\langle i, j \rangle$ as input. This algorithm looks for the stored encodings that involve the given edge and recursively searches for the tetrahdra that share the same edge (line 8-15). Similar to the R_{03} operation, this operation also has a $O(m)$ time complexity where m is the number of tetrahedra sharing at a given edge.

The last operation that belongs to the co-boundary relation retrieval operations family is the R_{23} that retrieves the tetrahedra that share a given face. The

```
algorithm R03 (t,i,tflags,tlist)
(1) if tflags[t] == 1 then
(2)    return
(3) endif
(4) tlist.append(t)
(5) tflags = 1
(6) numOfEncoding = countEncoding(t)
(7) for k = 0 to numOfEncoding
(8)    if vth bit of the encoding
(9)    getEncoding(k, t) is 1 then
(10)       t' ← getETetra(k, t)
(11)       v ← getV(t, i)
(12)       i' ← getPos(t', v)
(13)       R03 (t',i',tflags,tList)
(14)    endif
(15) endfor
(16) return
```

```
algorithm R13 (t,⟨i, j⟩,tflags,tlist)
(1) if tflags[t] == 1 then
(2)    return
(3) endif
(4) tlist.append(t)
(5) tflags = 1
(6) numOfEncoding = countEncoding(t)
(7) for k = 0 to numOfEncoding
(8)    if ith and jth bits of the encoding
(9)    getEncoding(k, t) are 1s then
(10)       t' ← getETetra(k, t)
(11)       ⟨u, v⟩ ← getE(t, i, j)
(12)       i' ← getPos(t', u)
(13)       j' ← getPos(t', v)
(14)       R03 (t',⟨i', j'⟩,tflags,tList)
(15)    endif
(16) endfor
(17) return
```

```
algorithm R23 (t,i,tlist)
(1) tlist.append(t)
(2) numOfEncoding = countEncoding(t)
(3) for k = 0 to numOfEncoding
(4)    if the encoding getEncoding(k, t) are
(5)    all 1s except for the ith bit then
(6)       tlist.append(getETetra(k, t))
(7)    endif
(8) endfor
(9) return
```

Figure 4-17. The algorithm for the R_{03} operation and the algorithm for the R_{13} operation.

implementation for this operation is trivial because every face is only shared by two tetrahedra and according to our definition for the spanning connectivity encodings, all encodings that involve a face are explicitly stored. We present the implementation of the operation R_{23} in Figure 4-17.

So far, we have introduced the algorithms for retrieving the co-boundary relations with $O(m)$ time complexity. The other operations, R_{00}, R_{11}, R_{22}, and R_{33}, belong to the adjacency relations family. Each of these operations can be implemented based on the corresponding co-boundary relations retrieval operations. For example, to retrieve the

vertices that are connected through an edge element to a given vertex (R_{00}), one only needs to retrieve all edges sharing the given vertex by calling the R_{01} operation, and the other end vertex points of the retrieved edges are the resulting vertices.

The implementation of the R_{11} operation is also based on the R_{01}. First, given an edge element $\langle u, v \rangle$, we can retrieve all edges in the mesh that share the vertex u and store them in a list $elist_1$, and retrieve all edges in the mesh that share the vertex v and store them in a list $elist_2$ by using the operation R_{01}. Then, the union of the two lists is the result of R_{11}. Similarly, we can implement the R_{22} operation based on the R_{12} operation. For each edge e_i of a given face element, we retrieve all faces sharing e_i by calling R_{12} operation, and we store them in a list $flist_i$. Then the union of all $flist_i$ gives the final result. To implement the R_{33} operation, we can retrieve all tetrahdra that share a face with the given tetrahedron by using the operation R_{23} to get the resulting tetrahedra for every face element of the given tetrahedron.

The time complexity of the adjacency relation retrieval operations are all $O(m)$ where m is the number of resulting elements.

4.2.4 The Evaluation and Experimental Study of the Connectivity Encoded Tetrahedral Mesh

In this section, we evaluate our CETM data structure from both the algorithmic perspective and the experimental perspective. We first analyze the storage complexity of our data structure and the time complexity of topological retrieval operators on our data structure, and compare them with the other existing data structures in Section 4.2.4.1 and Section 4.2.4.2.

Then we present some experimental study in Section 4.2.4.3 based on real datasets to demonstrate the efficiency of our CETM data structure in real world applications.

4.2.4.1 The Evaluation of the Storage Cost of our CETM Data Structure

So far, we have introduced our connectivity encoded tetrahedral mesh (CETM) data structure (Section 4.2.2), and the implementation of a complete set of topological relations retrieval operations (Section 4.2.3). We have also shown that the time complexity of the retrieval operations are linear to the number of elements retrieved, which is optimal. In this section, we analyze the storage cost of our proposed CETM data structure.

In our data structure, a total of five sequential arrays, which are the vertex array V, the tetrahedron-vertex array TV, the spanning connectivity encodings array E, the tetrahedron-encoding array TE, and the encoding-target tetrahedron array ET, are used (see Figure 4-16 for an example). Let n_t and n_v denote the number of tetrahedra in the mesh and the number of vertices in the mesh respectively. Further, we assume every vertex point is represented with 3 double values for x, y and z coordinates, and every reference that points to a vertex, an encoding, or a tetrahedron costs one integer. Then, the V array that records all vertices in the tetrahedral mesh costs $3n_v$ doubles, the TV array that records the references to the four conner points for each tetrahedron costs $4n_t$ integers, and the TE array that records the references to the encodings for each tetrahedron costs n_t integers. Let m denote the number of spanning connectivity encodings that are stored in the E array, then the E array that stores four bit encodings costs $4m$ bits and the ET array that stores the reference to the target tetrahedron for each encoding costs m integers. Therefore, a total of $3n_v$ doubles plus $5n_t + m$ integers plus $4m$ bits are stored for a tetrahedral mesh.

Now, the question is how big the m is. Since the m is the number of spanning connectivity encodings and the spanning connectivity encodings are a minimal set of encodings, m is expected to be a relative small value. By investigating the nature of the spanning connectivity encodings, we observe that the value of m relates to the number of shared faces, the number of non-manifold edges and the number of non-manifold

vertices. we can actually compute the precise value of m given a tetrahedral mesh. We explain this in details in the rest of this section.

In general, there are three types of connectivity encodings for the topological relations between two connected tetrahedra in a tetrahedral mesh, the encodings with three 1s indicating the sharing of a triangle face (adjacency), the encodings with two 1s indicating the sharing of an edge (incident), and the encodings of one 1 indicating the sharing of a vertex (incident). All encodings belong to one of the three types. So we investigate the number of spanning connectivity encodings for each of these types.

The first observation is that all face sharing connectivity encodings are spanning connectivity encodings, thus need to be physically stored. The proof of this observation is simple. According to the definition of derivable, all connectivity encodings with three 1s can not be derived from others. This is because that for any encoding c_{ij} that has three 1s, there does not exist a c_{ik} such that $c_{ik} \& c_{ij} = c_{ij}$. Otherwise, if there should exist such a c_{ik}, then either $c_{ik} = c_{ij}$, which means the kth tetrahedron shares a same face of the ith tetrahedron with the jth tetrahedron, or c_{ik} has four 1s. Both cases are not realistic. As a result, all encodings with three 1s must be included in the spanning connectivity encodings set. For example, in Figure 4-18A, the encodings that describe the adjacent relationship between t_1 and t_2, and the adjacent relationship between t_2 and t_3 need to be stored as spanning connectivity encodings.

The second observation is based on the investigation of the edge sharing based connectivity encodings. For any edge $w = \langle u, v \rangle$ in the tetrahedral mesh, we can use the edge star operation R_{13} to retrieve all tetrahedra incident at the edge w. We call two tetrahedra t_s and t_d 2-connected at w if there exists a 2-connected path $t_s, t_1, t_2, \ldots, t_k, t_d$ such that 1) t_s, t_d are adjacent to t_1, t_k respectively, 2) t_i is adjacent to t_{i+1} for $1 \leq i < k$, and 3) all tetrahedra on the path $t_s, t_1, t_2, \ldots, t_k, t_d$ are incident at the edge w. Further, we call an edge w the manifold edge if all tehtrahedra incident at w are 2-connected at w. Otherwise, if there exist a pair of tetrahedra incident at w between which we can not

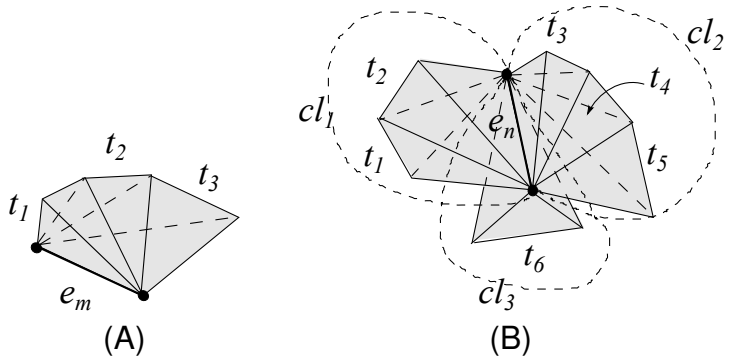


Figure 4-18. Examples of a manifold edge e_m and a non-manifold edge e_n in (A) and (B) respectively.

find a 2-connected path, then we call w a non-manifold edge. Figure 4-18A gives an example of a manifold edge e_m and Figure 4-18B gives an example of a non-manifold edge e_n. It is trivial to prove that all connectivity encodings that involve a manifold edge are not spanning connectivity encodings since they can always be derived from the face sharing connectivity encodings. For example, in Figure 4-18A, e_m is a manifold edge, and the connectivity encoding c_{13} between t_1 and t_3 that describes the sharing of the edge is not stored as a spanning connectivity encoding. According to the definition of derivable, c_{13} can be derived from c_{12} and c_{32}, where c_{12} is the face sharing encoding between t_1 and t_2, and c_{32} is the face sharing encoding between t_3 and t_2.

Hence, we only need to find out how many connectivity encodings are stored for a non-manifold edge. For all tetrahedra that are incident at a non-manifold edge w, we can group them into clusters so that any pair of tetrahedra belonging to the same cluster are 2-connected at w, and any pair of tetrahedra belonging to different clusters are not 2-connected at w. For example, in Figure 4-18B, e_n is a non-manifold edge since not all tetrahedra pair that are incident at e_n are 2-connected. Moreover, tetrahedra incident at e_n can be grouped into 3 clusters cl_1, cl_2 and cl_3, each contains one or more tetrahedra. As a result, for a non-manifold edge w that has clusters $cl_1, cl_2, \ldots, cl_k$, we only need to pick the representative tetrahedra $t_1, t_2, \ldots, t_k$, each from one cluster, and store the spanning connectivity encodings among these tetrahedra. We do not need to store the

encodings among tetrahedra within a same cluster since they can be derived from the face sharing connectivity encodings. Since $t_1, t_2, \ldots, t_k$ are from different clusters and are incident at a same edge w, the number of the spanning connectivity encodings among them must be $k-1$, no more and no less. We can prove this by constructing a graph G in which the nodes represent the tetrahedra $t_1, t_2, \ldots, t_k$. We create an edge in G that connects the nodes t_i and t_j if the encoding c_{ij} between t_i and t_j is counted as a spanning connectivity encoding. Therefore, if we have less than $k-1$ edges in graph G, then there must exist two nodes in G that are not connected, which means that the encoding between the two corresponding tetrahedra is not in the spanning connectivity encodings set and is not derivable from the current spanning connectivity encodings set. If we have more than $k-1$ edges, then there must exist a loop in the graph, which means that for any two directly connected nodes in the loop, they are also connected through other nodes. In other words, the spanning connectivity encoding is derivable from other encodings. Both these two cases violate the requirements of spanning connectivity encodings set. So we can conclude that for any manifold edge w_m, none of the encodings on w_m needs to be stored, and for any non-manifold edge w_n, we store a total of $2*(k-1)$ encodings as spanning connectivity encodings. We multiply $k-1$ by 2 because according to the definition of spanning connectivity encodings, if c_{ij} is in the spanning connectivity encodings set, then c_{ji} is also in the set. Figure 4-18B shows an example where only 4 spanning encodings need to be stored.

The third observation is based on the investigation of vertex sharing based encodings. Similar to the edge sharing case, we also distinguish vertices in a tetrahedral mesh as manifold vertices and non-manifold vertices. A vertex v is a manifold vertex if all tetrahedra incident at v are 1-connected at v, meaning that between any tetrahedra pair t_d and t_s there exists a 1-connected path $t_s, t_1, t_2, \ldots, t_k, t_d$ such that 1) t_s, t_d are adjacent to, or are incident at an edge to, t_1, t_k respectively, 2) t_i is adjacent to, or incident at an edge to, t_{i+1} for $1 \leq i < k$, and 3) all tetrahedra on the path

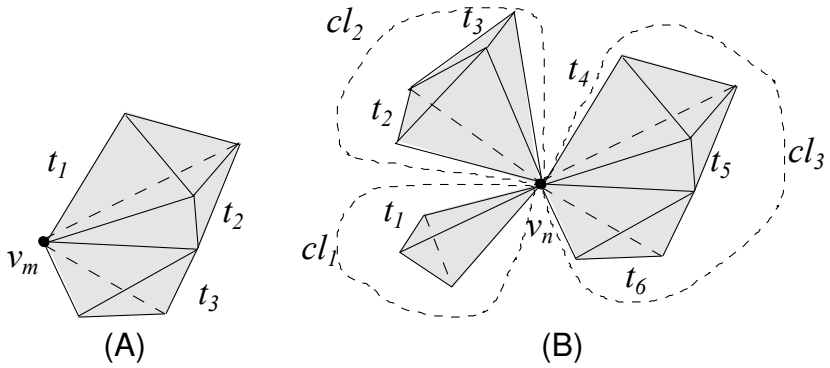


Figure 4-19. Examples of a manifold vertex v_m and a non-manifold vertex v_n in (A) and (B) respectively.

t_s, t_1, t_2, ..., t_k, t_d are incident at the vertex v. Otherwise, the vertex v is a non-manifold vertex. Figure 4-19A shows an example of a manifold vertex, and Figure 4-19B shows an example of a non-manifold vertex. The same cluster concept can also be applied here for all tetrahedra that are incident at a vertex v. In this case, then any tetrhadera pair in a same cluster are 1-connected at v, and two tetrahedra from different clusters are not 1-connected at v. A same strategy can be applied to prove the following conclusion that for any manifold edge w_m, none of the encodings on w_m needs to be stored, and for any non-manifold vertex v_n, we store a total of $2 * (k - 1)$ encodings as spanning connectivity encodings. In the example in Figure 4-19B, tetrahedra that are incident at vertex v_n can be grouped in to 3 clusters cl_1, cl_2 and cl_3. A total of 4 spanning encodings needs to be stored in this example.

Finally, we can compute the value for m, which is the number of spanning connectivity encodings stored for a tetrahedral mesh. Let C_e and C_v denote the total number of clusters around non-mandifold edges and the total number of clusters around non-manifold vertices respectively. Let s denote the number of shared faces. Then $m = 2(s + C_e + C_v)$. So the total storage cost of our CETM is about $5n_t + 2(s + C_e + C_v)$ integers, $3n_v$ doubles, and $8(s + C_e + C_v)$ bits.

4.2.4.2 The Comparisons of our CETM Data Structure with Others

In this section, we compare our approach with other existing work from both the storage cost and operation efficiency perspectives. In order to be fair, we only choose the data structures that at least offer optimal topological relations retrieval for manifold tetrahedral meshes. Data structures that may require less storage but do not support direct and random access to topological relations are not considered in this comparison. Examples are the compression strategies in [19, 47, 96], and the spatial index strategy in [].

We first present data structures that are called the Vertex Opposite Table (VOT) [12], the Sorted Opposite Table (SOT) [49], the Simplified Incident Graph (SIG) LDA04ACMSIGGRAPH, the Indexed Data Structure with Adjacencies (IA) [75, 77], and the Non-manifold Indexed data structure with Adjacencies (NMIA) [28]. The comparison among the last three data structures is presented in details in [29]. In this paper, we briefly introduce these data structures and show their storage requirements. We define the following notions that are used later. Let n_t, n_f, n_e, n_v denote the number of tetrahedra, faces, edges, and vertices in the tetrahedral mesh respectively. Let C_e and C_v denote the total number of clusters for non-manifold edges and non-manifold vertices respectively. Let s denote the number of shared faces. Further, we assume the size of a reference is the same as an integer value.

The idea behind the Vertex Opposite Table (VOT) data structure is to store the adjacent relationships among tetrahedra by storing only one opposite vertex for each corner of a tetrahedron. Therefore, the VOT data structure requires the storage of all vertices plus 8 references per tetrahedron that includes 4 references to the vertices of each tetrahedron and 4 references to the opposite vertices of the four conner points of a tetrahedron. Based on this data structure, the retrieval of adjacent tetrahedra has constant cost. However, since VOT only maintains the adjacent relationships among tetrahedra by using opposite vertices, it does not support the retrieval operations on

tetrahedral mesh that has non-manifold edges or non-manifold vertices. For example, VOT can retrieve all tetrahedra incident at edge e_m in Figure 4-18A, but it fails to retrieve all tetrahedra incident at edge e_n in Figure 4-18B. As a result, it requires the storage of $\mathbf{3n_v}$ ***doubles*** plus $\mathbf{8n_t}$ ***integers***.

An extension to VOT, called the *Sorted Opposite Table* (SOT), has been proposed in [49], which designs an even more compact data structure for holding tetrahedral mesh. By sorting the vertex list and by replacing the references to the vertices of each tetrahedron by a few services bits, it requires only 4 references and 9 service bits per tetrahedron. However, the trade off is the increase of the computational cost of accessing neighboring cells proportional to the valence of a common vertex. Moreover, since it is still based on the concept of storing adjacent relations by storing opposite vertices, it also does not support the retrieval of incident relations on a non-manifold edge or a non-manifold vertex. The total storage required for this data structure is $\mathbf{3n_v}$ ***doubles***, $\mathbf{4n_t}$ ***integers*** and $\mathbf{9n_t}$ ***service bits***.

The simplified incident graph data structure stores the coordinates of each vertex in the mesh (therefore requires $3n_v$ doubles where n_v is the number of vertices in the mesh). It also stores additional topological relations among tetrahedral mesh elements. Following relations are stored: 1) relation R_{32} that stores four faces on the boundary of each tetrahedron ($4n_t$ references); 2) relation R_{21} that stores three edges for each face ($3n_f$ references); 3) relation R_{10} that stores two vertices for each edge ($2n_e$ references); 4) relation R_{23} that stores adjacent relations (face sharing relations) for each tetrahedron ($4n_t$ references); 5) relation R_{12} that stores one representative face from each cluster incident at each non-manifold edge ($C_e + n_e - k$ references, where k is the number of non-manifold edges); 6) relation R_{01} that stores one representative edge from each cluster incident at each non-manifold vertex (C_v references); With a large amount of relations explicitly stored, this data structure is optimal in retrieval any topological

relations. As a result, a total of $\mathbf{3n_v}$ ***doubles*** plus ($\mathbf{8n_t + 3n_f + 3n_e + C_e + C_v - k}$) ***integers***.

The indexed data structure with adjacencies is a more compact data structure for tetrahedral meshes. Apart from storing the coordinates of vertices in the mesh, it only stores the R_{30} relation which stores four vertices on the boundary of each tetrahedron ($4n_t$ references), and the R_{33} relation that for each tetrahedron stores references to all its adjacent tetrahedra ($2s$ references). Similar to the vertex opposite table (VOT) data structure, it does not support the access to incident elements at a non-manifold edge or a non-manifold vertex. The total storage cost of this data structure is $\mathbf{3n_v}$ ***doubles*** plus ($\mathbf{4n_t + 2s}$) ***integers***.

The non-manifold indexed data structure with adjacencies is an extension to the above data structure. It handles not only a general tetrahedral mesh, but also any arbitrarily shaped objects that may even have tangling faces and tangle edges. The trade is then the need of additional storage. In this paper, we only investigate its storage requirements for a general tetrahedral mesh. In general, it stores vertices, tetrahdra vetex references, and a restricted set of topological relations. To be specific, there are a total of four topological relations explicitly stored, which are 1) R_{30} relation that stores references to the four conner vertices for each tetrahedron ($4n_t$ references); 2) R_{33} relation that stores $2n_t$ bits for flags (which indicate the number of adjacent tetrahedra for each tetrahedron) + ($2s$) integers for storing all adjacency relations + n_t references to the adjacent relations; 3) $R_{3,clusters}$ relations that stores cluster information for each of the six edges of a tetrahedron ($24n_t$ bits + $2C_e$ references) and the references to the cluster arrays (n_t); 4) $R_{0,clusters}$ relations that store $2C_v$ bits for flags + C_v references for the cluster arrays storing cluster representatives + n_v references keeping track of the length of each cluster array + n_v references to the array. More detailed description can be found in [28]. This data structure supports optimal retrieval operations for all topological relations. The total storage cost for a general

tetrahedral mesh possibly with non-manifold edges and non-manifold vertices is $\mathbf{3n_v}$ ***doubles***, $(\mathbf{6n_t + 2s + 2n_v + 2C_e + C_v})$ ***integers***, and $(\mathbf{26n_t + 2C_v})$ ***bits***.

We compare our CEMT data structure with these five data structures in Figure 4-20. The comparison table describes the storage cost of all the data structures for both manifold tetrahedral mesh and non-manifold tetrahedral mesh. For manifolds, the number of clusters at non-manifold edges (C_e) and the number of non-manifold clusters at non-manifold vertices (C_v) are zeros. The table also describes how the data structures support the incident relations retrieval operations R_{pq} where $p < q$. Among all data structures, the SOT is the most compact one. However, it does support optimal incident relations retrieval operations when non-manifold edges or non-manifold vertices exist in meshes. Data structures SIG, NMIA, and CETM all supports optimal incident relations retrieval operations on non-manifold elements of meshes. Since in practical applications, the number of shared faces is approximately the same as the number of tetrahedra ($s \approx n_t$), and it has been shown that $n_t \approx 6n_v$ [24], our data structure requires the least storage among the three data structures that support optimal incident relations retrieval operations on general tetrahedral meshes.

4.2.4.3 The Experimental Study on CETM

In the previous sections, we have evaluated the performance of the CETM data structure and have compared it with other popular data structures from the algorithmic perspective. In this section, we present some experimental study on the CETM data structure based on some benchmark datasets.

One available on-line benchmark for 3D shapes is the Princeton Shape Benchmark [84] which provides a repository of 3D models and software tools for evaluating shape-based retrieval and analysis algorithms. The benchmark contains a database of 3D polygonal models collected from the World Wide Web. For each 3D model, there is an Object File Format ($.off$) file with the polygonal geometry of the model, a model information file (e.g., the URL from where it came), and a JPEG image file with

	non-manifolds		manifolds	
	Storage Cost	R_{pq} $(p < q)$	Storage Cost	R_{pq} $(p < q)$
VOT	$3n_v$ D + $8n_t$ I	no	$3n_v$ D + $8n_t$ I	yes
SOT	$3n_v$ D + $4n_t$ I + $9n_t$ B	no	$3n_v$ D + $4n_t$ I + $9n_t$ B	yes
IA	$3n_v$ D + $(4n_t + 2s)$ I	no	$3n_v$ D + $(4n_t + 2s)$ I	yes
SIG	$3n_v$ D + $(8n_t + 3n_f + 3n_e + C_e + C_v - k)$ I	yes	$3n_v$ D + $(8n_t + 3n_f + 3n_e)$ I	yes
NMIA	$3n_v$ D + $(6n_t + 2s + 2n_v + 2C_e + C_v)$ I + $(26n_t + 2C_v)$ B	yes	$3n_v$ D + $(6n_t + 2s + 2n_v)$ I + $(26n_t + 2n_v)$ B	yes
CETM	$3n_v$ D + $(5n_t + 2s + 2C_e + 2C_v)$ I + $(8s + 8C_e + 8Cv)$ B	yes	$3n_v$ D + $(5n_t + 2s)$ I + $8s$ B	yes

Figure 4-20. The comparisons between our data structure with other five data structures for tetrahedral meshes, where n_t, n_f, n_e, n_v are the number of tetrahedra, the number of faces, the number of edges, and the number of vertices in the tetrahedral mesh, s is the number of shared faces in the mesh, and C_e, C_v are the total number of clusters at non-manifold edges and non-manifold vertices respectively. In this table, D represents double, I represents integer, and B represents bits.

a thumbnail view of the model. The benchmark contains a total of 1,814 models. In this article, we select some datasets from this repository and use them to validate our CETM data structure from both the storage perspective and the topological retrieval performance perspective. However, the 3D shapes provided in this repository are described with the boundary representation, and are stored in the *.off* file, the Object File Format, which is one of the popular file formats from the Geometry Center's *Geomview* package [76]. Thus they are not directly available for the input of our CETM data structure in the tetrahedralized form. So we apply a tool, the *TetGen* publicly available from [68], to first vaidate the 3D shapes in the datasets and convert the valid ones to

Table 4-1. The storage cost of CETM data structure for selected 3D shapes in the Princeton Shape Benchmark repository.

	*m*500	*m*471	*m*58	*m*531	*m*67
Vertices	355	2,329	2,149	14,219	103,910
Tetrahedra	1,109	7,335	8,455	53,817	464,736
Boundary Faces	612	4,398	3,016	23,100	111,526
NM-Edges	192	1,196	1,454	8,932	81,410
NM-Vertices	34	190	362	2,132	18,586
CE	4,050	26,328	32,620	203,232	1,847,414
CETM Size (byte)	48,925	321,072	367,466	2,332,140	20,101,923
.ele, *.node*, and *.neigh* Size (byte)	93,817	621,935	676,770	4,570,368	40,701,143
.mesh Size (byte)	127,180	814,715	897,698	5,776,764	46,939,479
.off Size (byte)	74,392	497,701	515,765	3,536,661	29,752,403

tetrahedral meshes. The tool generates constrained delaunay tetrahedralization (CDT) by implementing algorithms presented in [92, 93].

In order to evaluate the storage cost of the CETM data structure, we select the 3D shapes with the number of tetrahedra in the range from 1,109 to 464,736. Table 4-1 presents the storage cost of CETM for the selected 3D shapes in the Princeton Shape Benchmark repository.

In Table 4-1, *m*500, *m*471, *m*58, *m*531 and *m*67 are the ids of the 3D tetrahedral meshes that are generated from the corresponding 3D shapes in the Princeton Shape Benchmark repository by the TetGen tool. We have listed the number of vertices, the number of tetrahedra, the number of boundary faces (the triangle faces that are not shared), the number of non-manifold edges (NM-Edges), the number of non-manifold vertices (NM-Vertices) of a given tetrahedral mesh in the figure for all data samples. We also show the number of connectivity encodings that are stored in our CETM data structure and the size of the final CETM data structure in bytes for each data sample. In addition, we compare our CETM data structure with three file formats that are wildly used for storing tetrahedral meshes and storing general polyhedra. The *.node* file and the *.ele* file are used to store a list of vertex coordinates and a list of tetrahedra

Table 4-2. The cost of retrieving tetrahedra sharing the 3rd vertex of the 100th tetrahedron in CETM.

R03(100,3)	*m*500	*m*471	*m*58	*m*531	*m*67
Tetrahedra Visited	11	12	26	14	9
Tetrahedra retrieved	11	12	26	14	9
Execution Time (ms)	0.027	0.032	0.067	0.052	0.032

Table 4-3. The cost of retrieving tetrahedra sharing the 2nd and 3rd vertices of the 100th tetrahedron in CETM.

R03(100,2,3)	*m*500	*m*471	*m*58	*m*531	*m*67
Tetrahedra Visited	2	5	6	3	6
Tetrahedra retrieved	2	5	6	3	6
Execution Time (ms)	0.004	0.006	0.013	0.006	0.009

respectively. Several research applications, such as the Stellar program [59] and the *TetGen* tool [68], are developed based on this format. The *.neigh* file stores the topological relationships among mesh elements so that the topological relation can be retrieved efficiently. The *.mesh* files are used by *Medit*, a French program for visualizing scientific models. The program displays computation results of mechanics related to solids, fluids, thermics, electromagnetism, etc. We store the same tetrahedral mesh in different representations, and as a result, we can observe that our CETM data structures consumes the least storage.

Further, we perform experiments that retrieves the topological relations from a tetrahedral mesh. All other file formats do not support such operations except for the combination of *.ele*, *.node*, and *.neigh* files. As examples, we show the experimental results in Table 4-2 and Table 4-3.

We first retrieve all tetrahedra that share the 3rd vertex with the 100th tetrahedra in the mesh. Table 4-2 shows that the number of visited tetrahedra and the number of tetrahedra retrieved as result list are the same, and the time for retrieving the result is linear to the number of retrieved tetrahedra while is not affected by the size of the tetrahedral mesh at all. This means that in CETM the vertex start retrieval operation

is optimal. The same can be observed in Table 4-3 for the edge start relation retrieval operation.

4.3 A Comparison between The Two Approaches

The slice representation introduced presents a paradigm of designing data structures for spatial data types. It provides a clean and uniform data structure design and serves as the basis of efficient algorithms. For volume objects, the slice representation belongs to the boundary based volume representation approaches, which approximate the surface of a volume object with connected polygonal faces that encloses a subset of 3D space.

On the other hand, the TEN-based representation is one of the volume based representations that decomposes a volume object into smaller and simpler 3D elements, which are tetrahedra. We encode the interactions among tetrahedra and store them explicitly for efficient operations. An additional object index structure is also embedded in the volume representation for efficient data access. The volume intersection algorithm is given to show the benefit of our volume representation. In the future, we will explore more important operations which, for example, yield numerical values, or are predicates. Although different operations require different algorithms, a common paradigm exists for developing solutions based on our volume representation, that is to reduce the complex volume problems to simple tetrahedra problems.

Both representations are able to handle complex spatial objects like volumes with multiple components, and in both representations components are well organized so that efficient access is possible. However, since the slice representation aims at a general paradigm for all spatial data types, it is more suitable for cross data types operations like intersection between lines and volumes. Thus, the TEN-based representation is designed specifically for *volume* type so that the volume centric operations like volume computation and intersection between two volumes can be more efficient. Further, the TEN-based modeling strategy is widely applied in fields like GIS for

visualizing irregular objects like terrains, thus, 3D volume data in the form of a collection of tetrahedra is directly available. As a result, we design a TEN-based hierarchical volume representation in a database context for efficient querying and analysis purpose.

4.4 Detour: Computing the Cardinal Direction Development between Moving Points

As a detour from the cardinal directions between 3D spatial objects, we have proposed a modeling strategy for cardinal direction developments in Section 3.2.2.2, in which we have defined the development of cardinal directions over time as a sequence of temporally ordered and enduring cardinal directions. In this section, we propose our solution for computing such a cardinal direction development between two moving points from an algorithmic perspective. We base our solution on the slice representation of moving points, which represents the temporal development of a point with a sequence of timely ordered units called slices. We propose a three-phase solution for determining the developments of the directional relationships between two moving points. In a time-synchronized interval refinement phase, two moving points are refined by synchronizing their time intervals. As a result, each slice unit of the refined slice representation of the first moving point has a matching slice unit in the refined slice representation of the second moving point with the time interval. In the second phase, the slice unit direction evaluation phase, we present a strategy of computing cardinal directions between two slice units from both moving points. Finally, in the direction composition phase, the development of the cardinal direction is determined by composing cardinal directions computed from all slices pairs from both moving points.

The concept we have introduced in Section 3.2.2.2 serves as a specification for describing the changing cardinal directions between two moving points. However, issues like how to find common life time intervals and how to split them are left open. In this section, we overcome the issues from an algorithmic perspective. We first introduce the underlying data structure, called slice representation, for representing

moving points. Then we propose a three phase strategy including the time-synchronized interval refinement phase, the slice unit direction evaluation phase, and the direction composition phase.

4.4.1 The Slice Representation for Moving Points

Since we take the specification of the moving point data type in [34, 39] as our basis, we first review the representation of the moving point data type. According to the definition, the moving point date type describes the temporal development of a complex point object which may be a point cloud. However, we here only consider the simple moving point that involves exactly one single point. A slice representation technique is employed to represent a moving point object. The basic idea is to decompose its temporal development into fragments called "slices", where within each slice this development is described by a simple linear function. A slice of a single moving point is called a upoint, which is a pair of values (interval, unit-function). The interval value defines the time interval for which the unit is valid; the unit-function value contains a record (x_0, x_1, y_0, y_1) of coefficients representing the linear function $f(t)$=$(x_0 + x_1 t, y_0 + y_1 t)$, where t is a time variable. Such functions describe a linearly moving point. The time intervals of any two distinct slice units are disjoint; hence units can be totally ordered by time. More formally, let A be a single moving point representation, *interval* = *time* $\times$ *time*, *real4* = *real* $\times$ *real* $\times$ *real* $\times$ *real*, and *upoint* = *interval* $\times$ *real4*. Then A can be represented as an array of slice units ordered by time, that is, $A = \langle (I_1, c_1), (I_2, c_2), \ldots, (I_n, c_n) \rangle$ where for $1 \leq i \leq n$ holds that $I_i \in interval$ and $c_i \in real4$ contains the coefficients of a linear unit function f_i. Further, we require that $I_i < I_j$ holds for $1 \leq i < j \leq n$.

Figure 4-21 shows the slice representations of two single moving points A and B. In this example, t_i $(1 \leq i \leq 7)$ is a time instance and for $1 \leq i < j \leq 7$, $t_i < t_j$. The moving point A is decomposed into two slices with intervals $I_1^A = [t_2, t_4]$ and $I_2^A = [t_4, t_6]$. Let the function f_1^A with its coefficients c_1^A and the function f_2^A with its

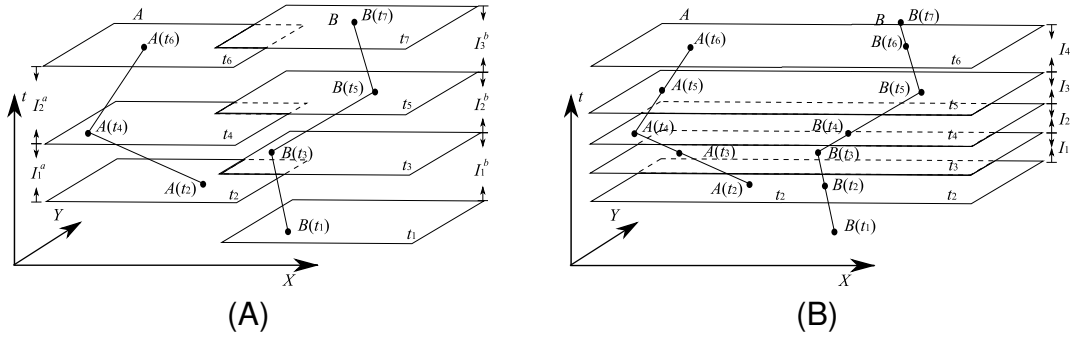


Figure 4-21. An example of the slice representations of two single moving points A and B (A), and the time-synchronized slice representation of two moving points A and B (B)

coefficients c_2^A describe the movement of A in the intervals I_1^A and I_2^A respectively. Then A is represented as $A = \langle (I_1^A, c_1^A), (I_2^A, c_2^A) \rangle$. The snapshots $f_1^A(t_2)$ and $f_1^A(t_4)$ of the moving point A at the times t_2 and t_4 are the start and end points of the first slice, and $f_2^A(t_4)$ and $f_2^A(t_6)$ are the start and end points of the second slice. Similarly, the moving point B can be represented as $B = \langle (I_1^B, c_1^B), (I_2^B, c_2^B), (I_3^B, c_3^B) \rangle$ where c_1^B, c_2^B, and c_3^B contain the coefficients of the three linear functions f_1^B, f_2^B, and f_3^B that describe the linear movement of B in its three slice units. If the function f_i^A or f_i^B that in a slice unit maps a time instant t to a point value in A or B is not important, we allow the notations $A(t)$ and $B(t)$ respectively to retrieve the location of a moving point A or B at the time instant t.

Further, we introduce a few basic operations for retrieving information from the slice representation, which will be used by our algorithm later for computing cardinal directions between moving points.

The first set of operations is provided for manipulating moving points. The *get_first_slice* operation retrieves the first slice unit in a slice sequence of a moving point, and sets the current position to 1. The *get_next_slice* operation returns the next slice unit of the current position in the sequence and increments the current position. The predicate *end_of_sequence* yields $true$ if the current position exceeds the end of the slice sequence. The operation *create_new* creates an empty $MPoint$ object with an

empty slice sequence. Finally, the operation *add_slice* adds a slice unit to the end of the slice sequence of a moving point.

The second set of operations is provided for accessing elements in a slice unit. The operation *get_interval* returns the time interval of a slice unit. The operation *get_unit_function* returns a record that represents the linear function of a slice unit. The *create_slice* operation creates a slice unit based on the provided time interval and the linear function.

Based on the slice representation and the basic operations, we are now ready to describe our strategy for computing the cardinal directions between two moving points.

4.4.2 The Time-synchronized Interval Refinement Phase

Since a slice is the smallest unit in the slice representation of moving points, we first consider the problem of computing cardinal directions between two moving point slices. According to our definitions in [21] the cardinal directions only make sense when the same time intervals are considered for both moving points. However, matching, i.e., equal, slice intervals can usually not be found in both moving points. For example, in Figure 4-21, the slice interval $I_1^A = [t_2, t_4]$ of A does not match any of the slice intervals of B. Although the slice interval $I_1^B = [t_1, t_3]$ of B overlaps with I_1^A, it also covers a sub-interval $[t_1, t_2]$ that is not part of I_1^A, which makes the two slices defined in I_1^A and I_1^B incomparable. Thus, in order to compute the cardinal directions between two moving point slices, a time-synchronized interval refinement for both moving points is necessary.

We introduce a linear algorithm $interval\_sync$ for synchronizing the intervals of both moving points. The input of the algorithm consists of two slice sequences $mp1$ and $mp2$ that represent the two original moving points, and two empty lists $nmp1$ and $nmp2$ that are used to store the two new interval refined moving points. The algorithm performs a parallel scan of the two original slice sequences, and computes the intersections between the time intervals from two moving points. Once an interval intersection is captured, two new slices associated with the interval intersection are created for

both moving points and are added to the new slice sequences of the two moving points. Let $I = [t_1, t_2]$ and $I' = [t'_1, t'_2]$ denote two time intervals, and let $lower\_than$ denote the predicate that checks the relationship between two intervals. Then we have $lower\_than(I, I') = true$ if and only if $t_2 < t'_2$. Further, let $intersection$ denote the function that computes the intersection of two time intervals, which returns $\emptyset$ if no intersection exists. We present the corresponding algorithm $interval\_sync$ in Figure 4-22.

As a result of the algorithm, we obtain two new slice sequences for the two moving points in which both operand objects are synchronized in the sense that for each unit in the first moving point there exists a matching unit in the second moving point with the same unit interval and vice versa. For example, after the time-synchronized interval refinement, the two slice representations of the moving points A and B in Figure 4-21 become $A = \langle (I_1, c_1^A), (I_2, c_1^A), (I_3, c_2^A), (I_4, c_2^A) \rangle$ and $B = \langle (I_1, c_1^B), (I_2, c_2^B), (I_3, c_2^B), (I_4, c_3^B) \rangle$, where the c_i^A with $i \in \{1, 2\}$ contain the coefficients of the linear unit functions f_i^A, the c_i^B with $i \in \{1, 2, 3\}$ contain the coefficients of the linear unit functions f_i^B, and $I_1 = intersection(I_1^A, I_1^B) = [t_2, t_3]$, $I_2 = intersection(I_1^A, I_2^B) = [t_3, t_4]$, $I_3 = intersection(I_2^A, I_2^B) = [t_4, t_5]$, and $I_4 = intersection(I_2^A, I_3^B) = [t_5, t_6]$.

Now we analyze the complexity of the algorithm for function $interval\_sync$. Assume that the first moving point mp_1 is composed of m slices, and the second moving point mp_2 is composed of n slices. Since a parallel scan of the slice sequences from two moving points is performed, the complexity is therefore $O(m + n)$ and the result contains at most $(m + n)$ intervals.

4.4.3 The Slice Unit Direction Evaluation Phase

From the first phase, the time-synchronized interval refinement phase, we obtain two refined slice sequences of both moving points that contain the same number of slice units with synchronized time intervals. In the second phase, we propose a solution for computing the cardinal directions between any pair of time-synchronized slice units.

```
method interval_sync (mp1, mp2,
                      nmp1, nmp2)
  s1 ← get_first_slice(mp1)
  s2 ← get_first_slice(mp2)
  while !end_of_sequence(mp1)
  and !end_of_sequence(mp2) do
    i1 ← get_interval(s1)
    i2 ← get_interval(s2)
    i ← intersection(i1, i2)
    if i ≠ ∅ then
      f1 ← get_unit_function(s1)
      f2 ← get_unit_function(s2)
      ns1 ← create_slice(i, f1)
      ns2 ← create_slice(i, f2)
      add_slice(nmp1, ns1)
      add_slice(nmp2, ns2)
    endif
    if lower_than(i1, i2) then
      s1 ← get_next_slice(mp1)
    else
      s2 ← get_next_slice(mp2)
    endif
  endwhile
end
```

```
 1  method compute_dir_dev(sl1, sl2)
 2    dev_list ← empty list
 3    s1 ← get_first_slice(sl1)
 4    s2 ← get_first_slice(sl2)
 5    slice_dir_list ← compute_slice_dir(s1,s2)
 6    append(dev_list, slice_dir_list)
 7    while not end_of_sequence(sl1)
 8    and not end_of_sequence(sl2) do
 9      (b, e) ← get_interval(s1)
10      s1 ← get_next_slice(sl1)
11      s2 ← get_next_slice(sl2)
12      (b_new, e_new) ← get_interval(s1)
13      if e < b_new then
14        append(dev_list, ⟨⊥⟩)
15      endif
16      slice_dir_list ← compute_slice_dir(s1,s2)
17      last_dir ← get_last_in_list(dev_list)
18      new_dir ← get_first_in_list(slice_dir_list)
19      if last_dir = new_dir then
20        remove_first(slice_dir_list)
21      endif
22      append(dev_list, slice_dir_list)
23    endwhile
24    return dev_list
25  end
```

Figure 4-22. The algorithms *interval_sync* and *compute_dir_dev*. The algorithm *interval_sync* refines the time-synchronized intervals for two moving points, and the algorithm *compute_dir_dev* computes cardinal direction developments between two moving points.

We adopt a two-step approach to computing the cardinal directions between two slice units. The first step is to construct a mapping and apply it to both slice units so that one of the slice units is mapped to a slice unit that consists of a point that does not change its location. We prove that the mapping is a cardinal direction preserving mapping that does not change the cardinal direction relationships between the two slice units. The second step is to determine the cardinal directions between the two mapped slice units.

The difficulty of computing cardinal directions between two slice units comes from the fact that the positions of the moving points change continuously in both slice units. A much simpler scenario is that only one slice unit consists of a moving point, whereas the other slice unit involves no movement. In this simpler case, the cardinal directions can be easily determined. Thus, the goal is to find a mapping that maps two slice units su_a and su_b to two new slice units su'_a and su'_b that satisfy the following two conditions: (i) su'_a and su'_b have the same cardinal directions as units su_a and su_b, that is, the mapping is a cardinal direction preserving mapping; (ii) either su'_a or su'_b does not involve movement.

In order to find such a mapping for two slice units, we first introduce a simple cardinal direction preserving mapping for static points. Let p and q denote two points with coordinates (x_p, y_p) and (x_q, y_q). Let $X(r)$ and $Y(r)$ denote the functions that return the x-coordinate and y-coordinate of a point r. We establish a simple translation mapping $M(r) = (X(r) - x_0, Y(r) - y_0)$, where x_0 and y_0 are two constant values. We show that the cardinal direction between p and q is preserved by applying such a mapping.

Lemma 14. *Given $p = (x_p, y_p)$, $q = (x_q, y_q)$, the mapping $M(r)$=($X(r) - x_0$, $Y(r) - y_0$), where r is a point and x_0 and y_0 are two constant values, and $p' = M(p)$ and $q' = M(q)$, we have $dir(p, q) = dir(p', q')$*

Proof. According to the definition in Section 3.2.2.2, the cardinal direction $dir(p, q)$ between two points p and q is based on the value of $X(p) - X(q)$ and the value of $Y(p) - Y(q)$. Since we have $X(p') - X(q') = X(p) - X(q)$ and $Y(p') - Y(q') = Y(p) - Y(q)$, we obtain $dir(p, q) = dir(p', q')$. □

In Figure 4-23A, two points p and q are mapped to p' and q', and the cardinal direction is preserved after the mapping, i.e., $dir(p, q) = dir(p', q') = NW$.

Now we are ready to define a cardinal direction preserving mapping for two slice units. Let su^A and su^B denote two slice units (upoint values) from the time-synchronized moving points A and B where $su^A = (I, c^A)$ and $su^B = (I, c^B)$ with $I \in interval$ and

$c^A, c^B \in real4$. Let f^A and f^B be the two corresponding linear unit functions with the coefficients from c^A and c^B respectively. We establish the following mapping M for a unit function $f \in \{f^A, f^B\}$: $M(f) = f - f^B$

We show in Lemma 15 that by mapping the unit functions of the slices su_A and su_B to two new unit functions, the cardinal directions between the slice units su_A and su_B are still preserved.

Lemma 15. *Let* $su^A = (I, c^A) \in$ upoint *and* $su^B = (I, c^B) \in$ upoint*, and let* f^A *and* f^B *be the corresponding linear unit functions with the coefficients from* $c^A = (x_0^A, x_1^A, y_0^A, y_1^A)$ *and* $c^B = (x_0^B, x_1^B, y_0^B, y_1^B)$ *respectively. We consider the mapping* $M(f) = f - f^B$*, where* f *is a linear unit function, and the translated* upoint *values* $su_t^A = (I, c_t^A)$ *and* $su_t^B = (I, c_t^B)$ *where* c_t^A *and* c_t^B *contain the coefficients of* $M(f^A)$ *and* $M(f^B)$ *respectively. Then, the cardinal directions between the slice units* su_t^A *and* su_t^B *are the same as the cardinal directions between the slice units* su^A *and* su^B*.*

Proof. Let $I = [t_1, t_2]$, $f^A(t) = (x_0^A + x_1^A t, y_0^A + y_1^A t)$, and $f^B(t) = (x_0^B + x_1^B t, y_0^B + y_1^B t)$. Then we have $M(f^A) = (x_0^A - x_0^B + (x_1^A - x_1^B)t, y_0^A - y_0^B + (y_1^A - y_1^B)t)$ and $M(f^B) = (0, 0)$. Assume there exists a time t_0 ($t_1 \leq t_0 \leq t_2$) such that $dir(f^A(t_0), f^B(t_0)) \neq dir(M(f^A)(t_0), M(f^B)(t_0))$. Let $x^B = x_0^B + x_1^B t_0$ and $y^B = y_0^B + y_1^B t_0$ denote two constant values. Since $M(f^A)(t_0) = (x_0^A - x_0^B + (x_1^A - x_1^B)t_0, y_0^A - y_0^B + (y_1^A - y_1^B)t_0)$ and $M(f^B)(t_0) = (0, 0)$, we have $M(f^A)(t_0) = (X(f^A(t_0)) - x^B, Y(f^A(t_0)) - y^B)$ and $M(f^B)(t_0) = (X(f^B(t_0)) - x^B, Y(f^B(t_0)) - y^B)$. This matches the cardinal direction preserving mapping function $M(r) = (X(r) - x_0, Y(r) - y_0)$. Thus, the assumption $dir(f^A(t_0), f^B(t_0)) \neq dir(M(f^A)(t_0), M(f^B)(t_0))$ contradicts to Lemma 14. □

After applying the cardinal direction preserving mapping $M(f)$ to both unit functions f^A and f^B, we now obtain two new unit functions f'_a and f'_b as follows:

$$g^A(t) = M(f^A)(t) = (x_0^A - x_0^B + (x_1^A - x_1^B)t, y_0^A - y_0^B + (y_1^A - y_1^B)t)$$
$$g^B(t) = M(f^B)(t) = (0, 0)$$

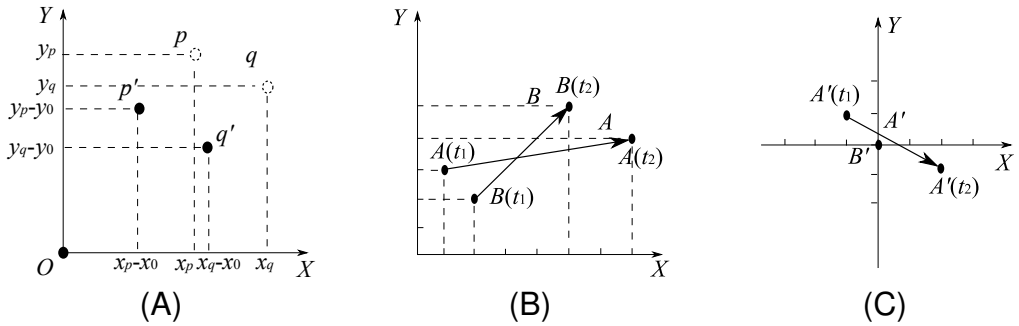


Figure 4-23. A simple cardinal direction preserving mapping from p, q to p',q' (A), and a cardinal direction preserving mapping from slice unit A, B (B) to A',B' (C).

The unit function g^A describes a linear movement in the unit interval, while the unit function g^B describes a static point that holds its position during the entire unit interval. In other words, su^A is mapped to a new slice unit su_t^A which has a linear movement, and su^B is mapped to a new slice unit su_t^B that has no movement during the unit interval. Figure 4-23 shows an example of mapping the slice units A and B to slice units A' and B'. In this example, $A = [I, c^A]$ and $B = [I, c^B]$ where $I = [1, 2]$, c^A and c^B contain the coefficients of the two unit functions f^A and f^B respectively, $f^A(t) = (-5 + 6t, 2 + t)$ and $f^B(t) = (-1 + 3t, -1 + 3t)$. Thus, $A(t_1) = f^A(1) = (1, 3)$, $A(t_2) = f^A(2) = (7, 4)$, $B(t_1) = f^B(1) = (2, 2)$, and $B(t_2) = f^B(2) = (5, 5)$. After applying the mapping, we obtain $g^A(t) = (-4 + 3t, 3 - 2t)$ and $g^B(t) = (0, 0)$. Thus, $A'(t_1) = g^A(1) = (-1, 1)$, $A'(t_2) = g^A(2) = (2, -1)$, and $B'(t_1) = B'(t_2) = (0, 0)$.

So far, we have managed to reduce the problem of computing the cardinal directions between two moving slice units to the problem of computing the cardinal directions between one moving slice unit and one static slice unit. The second step is to compute the cardinal directions between su_t^A and su_t^B.

Since su_t^B is located constantly at $(0, 0)$ during the time interval and since the trajectory of su_t^A is a linear function with respect to time t, we apply the projection based approach to determining the cardinal directions. The idea is to take su_t^B as the reference point and to create partitions by using the x- and y-coordinate axes. Then we project the slice unit su_t^A to the xy-plane, and the cardinal directions are determined by the partitions

that its trajectory intersects. Finally, the cardinal directions are ordered according to the time when they occurred and are stored into a list. For example, the cardinal directions between A' and B' in Figure 4-23B are *NW*, N, *NE*, E, and *SE*.

4.4.4 The Direction Composition Phase

Finally, in the direction composition phase, we iterate through all slice units, compose all cardinal directions that have been detected in slice units, and form a complete cardinal direction list in the temporal order. Further, we remove duplicates between consecutive cardinal directions.

We introduce the linear algorithm $compute\_dir\_dev$ in Figure 4-22 for computing the final cardinal direction development (line 24) between two synchronized moving points. The input of the algorithm consists of two lists of slices $sl1$ and $sl2$ (line 1) that stem from the time-synchronized interval refinement phase. Since the two slice lists are guaranteed to have the same length, the algorithm takes a slice from each list (lines 3, 4, 10 and 11), determines the cardinal directions for each pair of slices (lines 5 and 16), which have the same unit interval, and traverses both lists in parallel (lines 7 and 8). For two consecutive pairs of slices, we have to check whether the slice intervals are adjacent (lines 9, 12, and 13). If this is not the case, we add the list with the single element $\perp$ to the global list *dev_list* in order to indicate that the cardinal direction development is undefined between two consecutive slice intervals (lines 13 to 15).

For each pair of slices, the function *compute_slice_dir* determines their cardinal directions according to the strategy discussed in Section 4.4.3 (lines 5 and 16). We maintain a list *slice_dir_list* to keep these newly computed cardinal directions from the current slice pair and compare its first cardinal direction with the last cardinal direction that has been computed from the last slice pair and is stored in the global list *dev_list* (lines 17 to 19). If both cardinal directions are the same, the first cardinal direction from the list *slice_dir_list* is removed in order to avoid duplicates (lines 19 to 21). The newly

computed cardinal directions in the list *slice_dir_list* are added to the global list *dev_list* (lines 6 and 22).

The algorithm $compute\_dir\_dev$ deploys a number of auxiliary list functions. The function *get_first_in_list* returns the first element in a list. The function *get_last_in_list* returns the last element in a list. The function append adds a list given as its second argument to the end of another list given as its first argument. The function *remove_first* removes the first element from a list.

Now we analyze the complexity of the algorithm for function $compute\_dir\_dev$. Assume that the first moving point mp_1 consists of m slices, and the second moving point mp_2 consists of n slices. The inputs of the function $compute\_dir\_dev$ are two lists of slices generated from the time-synchronized interval refinement phase, thus each list contains at most $m + n$ slices. The function $compute\_dir\_dev$ iterate through all slices in both list and compose the cardinal directions computed. So the time complexity is $O(m + n)$.

CHAPTER 5
INTEGRATION AND IMPLEMENTATION OF THREE-DIMENSIONAL DATA IN DATABASES

Finally, with the *Spatial Algebra 3D at the Abstract Level* (*SPAL3D-A*) introduced in Chapter 3, and the *Spatial Algebra 3D at the Discrete Level* (*SPAL3D-D*) introduced in Chapter 4, we now explore in details how our 3D data model can be integrated in databases. There are two main tasks involved in the integration process, the integration of 3D data in database tables and the integration of operations and predicates in SQL language.

The data representations for 3D objects are usually structured and complex, and they supports operations like insert, update, and search on its components. Traditional database management systems store and manage either simple alphanumerical data or unstructured byte level data, thus are not directly suitable for storing 3D data. To bridge this gap, in this chapter, we propose a generic low level framework for handling large, structured, complex objects like 3D volumes in a database context, and show that it supports all required operations like insert, update and search at the component level. It is necessary to emphasis that the goal is not to design it only for 3D spatial types, but to design it as a generic framework that is beneficial to the implementation of all data types that deal with large, structured and complex objects, examples other than 3D volumes are the 2D regions and moving hurricanes.

The second task is to properly design a collection of 3D operations and 3D predicates, and integrate them into the SQL language so that they become available to the user through the well know textual SQL language.

5.1 Integration of 3D Spatial Data Types in Databases

Many fields in computer science are increasingly confronted with the problem of handling large, variable-length, highly structured, complex application objects and enabling their storage, retrieval, and update by application programs in a user-friendly, efficient, and high-level manner. Examples of such objects include biological sequence

data, spatial data, spatiotemporal data, multimedia data, and image data, just to name a few. Traditional database management systems (DBMS) are well suited to store and manage large, unstructured alphanumeric data. However, storing and manipulating large, structured application objects at the low byte level as well as providing operations on them are hardly supported. Binary large objects (*BLOBs*) are the only means to store such objects. However, BLOBs represent them as low-level, binary strings and do not preserve their structure. As a result, this database solution turns out to be unsatisfactory. Thus, we propose a novel two-step solution to manage and query application objects within databases.

Section 5.1.1 describes relevant research related to the storage and management of structured objects. In Section 5.1.2, we describe the applications that involve large structured application objects, the existing approaches to handling them, and our approach to dealing with structured objects in a database context. In Section 5.1.3, we introduce the concept of type structure specification and the iBLOB framework. Finally, we evaluate the iBLOB performance based on real datasets.

5.1.1 Existing Approaches for Handling Structured Objects

The need for extensibility in databases, in general, and for new data types in databases [95], in particular, has been the topic of extensive research from the late eighties. In this section, we review work related to the storage and management of structured large application objects. The four main approaches can be subdivided into specialized file formats, new DBMS prototypes, traditional relational DBMS, and object-oriented extensibility mechanisms in DBMS.

The specialized file formats can be further categorized into text formats and binary formats [69]. Text formats organize data as a stream of Unicode characters whereas binary formats store numbers in “native” formats. XML [15] is a universal standard text data format primarily meant for data exchange. A critical issue with all text data formats is that they make the data structure visible and that one cannot randomly access specific

subcomponent data in the middle of the file. The whole XML file has to be loaded into the main memory to extract the data portion of interest. Moreover, the methods used to define the legal structure for a XML document such as Document Type Definition (DTD) and XML Schema Definition (XSD) have several shortcomings. DTD lacks support for datatypes and inheritance, while XSD is really over-verbose and unintuitive when defining complex hierarchical objects. On the other hand, binary data formats like NetCDF [69, 85] and HDF [1, 69] support random access of subcomponent data. But updating an existing structure is not explicitly supported in both formats. Further, since HDF stores a large amount of internal structural specifications, the size of a HDF file is considerably larger than a flat storage format. Further, these file formats do not benefit from DBMS properties such as transactions, concurrency control, and recovery.

The second approach to storing large objects is the development of new DBMS prototypes as standalone data management solutions. These include systems such as *BSSS* [53], *DASDBS*, [87], *EOS* [11], *Exodus* [18], *Genesis* [7], and *Starburst* [51]. These systems operate on variable-length, uninterpreted byte sequences and offer low-level byte range operations for insertion, deletion, and modification. However, these systems do not manage structural information of large application objects and are hence unable to provide random access to object components.

The third approach taken to store large objects is the use of tables and BLOBs in traditional object-relational database management systems. Any hierarchical structure within an object can be incorporated in tables using a separate attribute column that cross-references tuples with their primary keys. Some database such as Oracle even support hierarchical SQL queries on such tables. However, the drawback of this method is that the querying becomes unintuitive and has to be supported by complex procedural language functions inside the database. Further, these queries are slow because of the need of multiple joins between tables. Binary Large OBjects (BLOBs) provide another means to store large objects in databases. However, this is a mechanism for storing

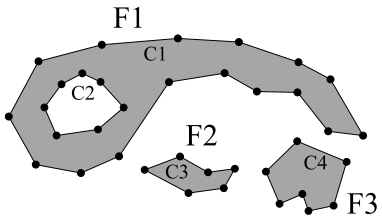


Figure 5-1. A region object as an example of a complex, structured application object. It contains the faces F1, F2, and F3, which consist of the cycles C1 and C2 for F1, C3 for F2, and C4 for F3.

unstructured, binary data. Hence, the entire BLOB has to be loaded into main memory each time for processing purposes.

The fourth approach to storing large objects is the use of object-oriented extension mechanisms in databases. Most popular DBMS support the CREATE TYPE construct to create user-defined data types. However, the type constructors provided (like array constructors) do not allow to create large and variable-length application objects.

5.1.2 Problems with Handling Structured Objects in Databases and Our Approach

Application objects like DNA structures, 3D buildings, and spatial regions are complex, highly structured, and of variable representation length. The desired operations on the application objects usually involve high complexity, long execution time and large memory. For example, region objects are complex application objects that are frequently used in GIS applications. As shown in Figure 5-1, a region object consists of components called faces, and faces are enclosed by cycles. Each cycle is a closed sequence of connected segments. Applications that deal with regions might be interested in numeric operations that compute the area, the perimeter and the number of faces of a region. They might also be interested in geometric operations that compute the intersection, union, and difference of two regions. Many more operations on regions are relevant to applications that work with maps and images. In any case, the implementation of an operation requires easy access to components of structured

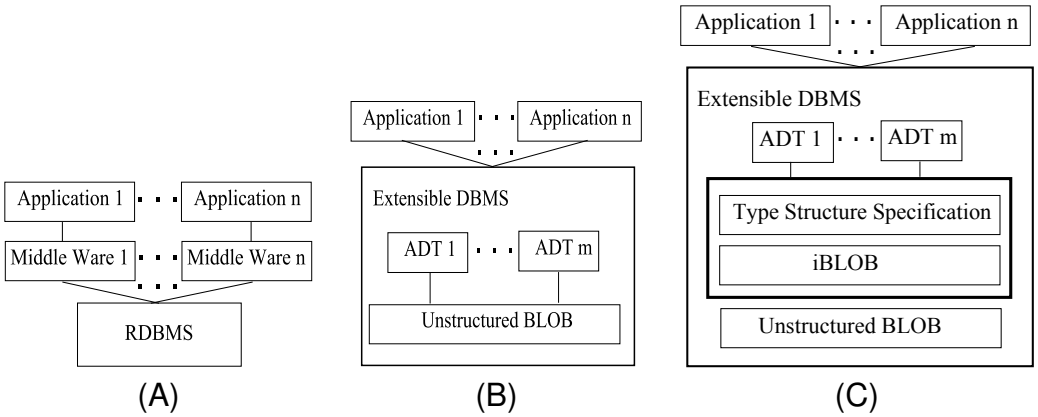


Figure 5-2. Three modeling architectures. The layered architecture (A), the integrated architecture (B) and our solution (C).

objects (e.g., segments, cycles, and faces of a region) that uses less memory and runs in less time.

Since database systems provide built-in advanced features like the SQL query language, transaction control, and security, handling complex objects in a database context is an expedient strategy. Most approaches are built upon two important architectures that enable database support for applications involving complex application objects.

Early approaches apply a layered architecture as shown in Figure 5-2A, in which a middleware that handles complex application objects is clearly separated from the application front-end that provides services and analysis methods to its users. In this architecture, only the underlying primitive data are physically stored in traditional RDBMS tables. The knowledge about the structure of complex objects is maintained in the middleware. It is the responsibility of the middleware to load the primitive data from the underlying database tables, to reconstruct complex objects from the primitive data, and to provide operations on complex objects. The underlying DBMS in the layered architecture does not understand the semantics of the complex data stored. In this sense, the database is of limited value, and the burden is on the application developer to

implement a middleware for handling complex objects. This complicates and slows down the application development process.

A largely accepted approach is to model and implement complex data as abstract data types (*ADTs*) in a type system, or algebra, which is then embeded into an extensible DBMS and its query language. This approach employs an integrated architecture (Figure 5-2B), where the applications directly interact with the extended database system, and use the ADTs as attribute data types in a database schema. Some commercial database vendors like Oracle and Postgres have included some ADTs like spatial data types as built-in data types in their database products. Extensible DBMS provides users the interfaces for implementing their own ADT so that all types of applications can be supported. Since the only available data structure for storing complex objects with variable length is BLOB, the implementations of ADTs for complex objects are generally based on BLOBs. The implementation of an abstract data type involves three tasks, the design of binary representation, the implementation of component retrieval and update, and the implementation of high level operations and predicates.

The integrated architecture has obvious advantages. It transfers the burden of handling complex objects from the application developer to databases. Once abstract data types are designed and integrated into a database context, applications that deal with complex objects become standard database applications, which require no special treatment. This simplifies and speeds up the development process for complex applications. However, the drawback of this approach is that ADTs for structured application objects rely on the unstructured BLOB type, which provides only byte level operations that complicate, or even foil, the implementation of component retrieval and update. Byte manipulation is a redundant and tedious task for type system implementers who want to implement a high-level type system because they want to focus on the design of the data types and the algorithms for the high-level operations and predicates.

In this paper, we propose a new concept that extends the integrated architecture approach, provides the type system implementers with a high level access to complex objects, and is capable of handling any structured application objects.
In our concept, we apply the integrated architecture approach and extend it with a generalized framework (Figure 5-2C) that consists of two components, the type structure specification (Section 5.1.3.1) and the intelligent BLOB concept (Section 5.1.3.2). The type structure specification consists of algebraic expressions that are used by type system implementers to specify the internal hierarchy of the abstract data type. It is later used as the meta data for the intelligent BLOB to identify the semantic meaning of each structure component. Further, as part of the type structure specification we provide a set of high-level functions as interfaces for type system implementers to create, access, or manipulate data at the component level. To support the corresponding interfaces, we propose a generic storage method called intelligent BLOB (*iBLOB*), which is a binary array whose implementation is based on the BLOB type and which maintains hierarchical information. It is "intelligent" because, unlike BLOBs, it understands the structure of the object stored and supports fast access, insertion and update to components at any level in the object hierarchy.

The type structure specification in the framework provides an abstract view of the application object which hides the implementation details of the underlying data structure. The underlying intelligent BLOBs ensure a generic storage solution for any kinds of structured application objects, and enable the implementation of the high-level interfaces provided by the type structure specification. Therefore, the type structure specification and the concept of intelligent BLOBs together enable an easy implementation for abstract data types. type system implementers can be released from the task of interpreting the logical semantics of binary unstructured data, and the component level access is natively supported by the underlying iBLOB.

5.1.3 iBlob: A Blob-based Generic Type System Implementation

In this section, we present a novel, generic model for complex object management that focuses on providing the required functionality to address the data management, generality, abstraction, and update problems. We first propose a generalized method, named type structure specification, for representing and interpreting the structure of application objects. This specification provides an interface for the ADT implementer to describe the structure of complex objects at the conceptual level. Based on this specification, we employ a generalized framework, called intelligent binary large objects (*iBLOBs*), for the efficient and high-level storage, retrieval, and update of hierarchically structured complex objects in databases. iBLOBs store complex objects by utilizing the unstructured storage capabilities of DBMS and provide component-wise access to them. In this sense, they serve as a communication bridge between the high-level abstract type system and the low-level binary storage. This framework is based on two orthogonal concepts called structured index and sequence index. A structured index facilitates the preservation of the structural composition of application objects in unstructured BLOB storage. A sequence index is a mechanism that permits full support of random updates in a BLOB environment.

5.1.3.1 Type Structure Specifications

The structures of different application objects can vary. Examples are the structure of a region (Figure 5-1) and the structure of a book. We aim at developing a generic platform that accommodates all kinds of hierarchical structures. Thus, the first step is to explore and extract the common properties of all structured objects. Unsurprisingly, the hierarchy of a structured object can always be represented as a tree. Figure 5-3A shows the tree structure of a region object. In the figure, face[], holeCycle[], and segment[] represent a list of faces, a list of hole cycles and a list of segments respectively. In the tree representation, the root node represents the structured object itself, and each child node represents a component named sub-object. A sub-object can further have

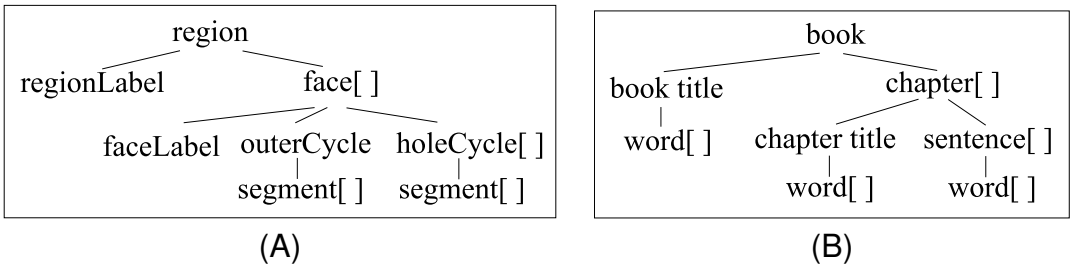


Figure 5-3. The hierarchical structure of a region object and the hierarchical structure of a *book* object.

a structure, which is represented in a sub-tree rooted with that sub-object node. For example, a region object in Figure 5-3A consists of a label component and a list of face components. Each face in the face list is also a structured object that contains a face label, an outer cycle, and a list of hole cycles, where both the outer cycle and the hole cycles are formed by segments lists. Similarly, the structure of a book can also be represented as a tree (Figure 5-3B).

Further, we observe that two types of sub-objects can be distinguished called structured objects and base objects. Structured objects consist of sub-objects, and base objects are the smallest units that have no further inner structure. In a tree representation, each leaf node is a base object while internal nodes represent structured objects.

A tree representation is a useful tool to describe hierarchical information at a conceptual level. However, to give a more precise description and to make it understandable to computers, a formal specification would be more appropriate. Therefore, we propose a generic type structure specification as an alternative of the tree representation for describing the hierarchical structure of application objects.

We first introduce the concept of structure expressions. Structure expressions define the hierarchy of a structured object. A structure expression is composed of structure tags (*TAGs*) and structure tag lists (*TAGLISTs*). A structure tag (TAG) provides the declaration for a single component of a structured object, whereas a structure tag

list (TAGLIST) provides the declaration for a list of components that have the same structure. The declaration of a TAG, named tag declaration, is $\langle NAME : TYPE \rangle$, where $NAME$ is the identifier of the tag and the value of $TYPE$ is either SO, which is a flag that indicates a structured object, or BO, which is a flag that indicates a base object. An example of a structured object tag is $\langle region : SO \rangle$, and $\langle segment : BO \rangle$ is an example of a base object tag. We first define a set of terminals that will be used in structure expressions as constants. Then, we show the syntax of structure expressions.

$$
\begin{aligned}
&\textit{Terminal Set } S = \{:=,\ \langle,\ \rangle,\ |,\ [,\ ],\ SO,\ BO,\ :\} \\
&\textit{Expression} \quad ::= TAG := \langle TAG \mid TAGLIST \rangle^{+}; \\
&TAGLIST \quad ::= TAG\,[\,] \\
&TAG \quad ::= \langle NAME : TYPE \rangle \\
&TYPE \quad ::= \langle SO \mid BO \rangle \\
&NAME \quad ::= IDENTIFIER
\end{aligned}
$$

In the region example, we can define the structure of a region object with the following expression: $\langle region : SO \rangle := \langle regionLabel : BO \rangle \langle face : SO \rangle [\,]$. In the expression, the left side of $:=$ gives the tag declaration of a region object and the right side of $:=$ gives the tag declarations of its components, in this case, the region label and the face list. Thus, we say the region object is defined by this structure expression.

With structure expressions, the type system implementer can recursively define the structure of structured sub-objects until no structured sub-objects are left undefined. A list of structure expressions then forms a specification. We call a specification that consists of structure expressions and is organized following some rules a type structure specification (*TSS*) for an abstract data type. Three rules are designed to ensure the correctness and completeness of a type structure specification when writing structure expressions: (1) the first structure expression in a TSS must be the expression that defines the abstract data type itself (correctness); (2) every structured object in a TSS has to be defined with one and only one structure expression (completeness and

uniqueness); (3) none of the base objects in a TSS is defined (correctness). By following these rules, the type system implementer can write one type structure specification for each abstract data type. Further, it is not difficult to observe that the conversion between a tree representation and a type structure specification is simple. The root node in a tree maps to the first structure expression in the TSS. Since all internal nodes are structured sub-objects and leaf nodes are base sub-objects, each internal node has exactly one corresponding structure expression in the TSS, and leaf nodes require no structure expressions. The type structure specification of the abstract data type region corresponding to the tree structure in Figure 5-3A is as follows:

$$\langle region : SO\rangle := \langle regionLabel : BO\rangle\langle face : SO\rangle[\,];$$
$$\langle face : SO\rangle := \langle faceLabel : BO\rangle\langle outerCycle : SO\rangle\langle holeCycle : SO\rangle[\,];$$
$$\langle outerCycle : SO\rangle := \langle segment : BO\rangle[\,];$$
$$\langle holeCycle : SO\rangle := \langle segment : BO\rangle[\,];$$

The next step after specifying the structure is to create and store the application object into the database. The TSS provides a workable interface for the type system implementer to create, access and navigate through the object. This higher-level interface is the abstraction of the iBLOB interface. This abstraction along with the specification, frees the type system implementer from understanding the underlying data type iBLOB that is used for finally representing the application object in the database. Navigating through the structure of the object is done by specifying a path from the root to the node by a string using the dot-notation. For example, to point to the first segment of the outer cycle of the third face of a region object can be specified by the string region.face[3].outerCycle.segment[1]. A component number (e.g., first segment, third face) is determined by the temporal order when a component was inserted. An important point to mention is that the structural validity of a path (e.g., whether an outer cycle is a subcomponent of a face) can be verified by parsing the TSS. However, the

existence of a third face can only be detected during runtime. The set of operators which are defined by the interface are given below:

$$
\begin{aligned}
&create && :\rightarrow SO\\
&get && : path \rightarrow BO[]\\
&set && : path \rightarrow bool\\
&set && : path \times char* \rightarrow bool\\
&baseObjectCount && : path \rightarrow int\\
&subObjectCount && : path \rightarrow int
\end{aligned}
$$

An application object can be created by the operator *create*() which generates an empty application object. The operator $get(p)$ returns all base objects at leaf nodes under the node specified by any valid path p. Since no data types are defined for the structured objects in intermediate nodes, these objects are not accessible, and paths to them are undefined. Hence, paths to intermediate nodes are interpreted differently in the sense that the operator $get(p)$ recursively identifies and returns all base objects under p. The operator $set(p)$ creates an intermediate component. The operator $set(p, s)$ inserts a base object given as a character string s at the location specified by the path p. The last two operators $baseObjectCount(p)$ and $subObjectCount(p)$ return the number of base objects and the number of sub-objects under a node specified by the path p. As an example, for a region object with one face that contains an outer cycle with three segments, the corresponding code for creating the region object is given below:

$$
\begin{aligned}
&region\ r = create();\ r.set(region.regionLabel, "MyRegion");\\
&r.set(region.face[1]);\ r.set(region.face[1].faceLabel, "Face1");\\
&r.set(region.face[1].outerCycle);\\
&r.set(region.face[1].outerCycle.segment[1], seg1);\\
&r.set(region.face[1].outerCycle.segment[2], seg2);\\
&r.set(region.face[1].outerCycle.segment[2], seg3);
\end{aligned}
$$

The first line of the code shows how the type system implementer can create a region object based on the specified type structure specification. The second line

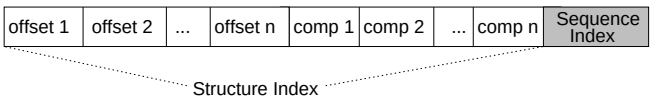


Figure 5-4. Illustration of an iBLOB object consisting of a structure index and a sequence index.

creates the first face and the third line its outer cycle as intermediate components. The following three lines store the three segments *seg*1, *seg*2, *seg*3 as components of the outer cycle.

5.1.3.2 Intelligent Binary Large Objects (iBLOB)

In this section, we present the conceptual framework for a new database data type called *iBLOB* for *Intelligent Binary Large Objects*. This type enhances the functionality of traditional binary large objects (BLOBs) in database systems. BLOBs serve currently as the only means to store large objects in DBMS. However, they do not preserve the structure of application objects and do not provide access, update and query functionality for the sub-components of large objects. *iBLOB*s help to smartly extend traditional BLOBs by preserving the object structure internally and providing application-friendly access interfaces to the object components. All this is achieved while maintaining low level access to data and extending existing database systems using object-oriented constructs and abstract data types (*ADTs*).

The iBLOB framework consists of two main sections called the structure index and the sequence index (Figure 5-4). The first section contains the structure index which helps us represent the object structure as well as the base data. The second section contains the sequence index that dictates the sequential organization of object fragments and preserves it under updates. Since the underlying storage structure of an iBLOB is provided through a BLOB, which is available in most DBMSs, the iBLOB data type can be registered as a user-defined data type and be used in SQL.

- iBLOB structure index: preserving structure in unstructured storage

A structure index is a mechanism that allows an arbitrary hierarchical structure to be represented and stored in an unstructured storage medium. It consists of two components for, first, the representation of the structure of the data and, second, the actual data themselves. The structural component is used as a reference to access the data's structural hierarchy. The mechanism is not intended to enforce constraints on the data within it; thus, it has no knowledge of the semantics of the data upon which it is imposed. This concept considers hierarchically structured objects as consisting of a number of variable-length sub-objects where each sub-object can either be a structured object or a base object. Within each structured object, its sub-objects reside in sequentially numbered slots. The leaves of the structure hierarchy contain base objects.

To illustrate the concept of a structure index, we show an example how to store a spatial region object with a specific structure in a database. A region data type may be described by a hierarchical structure as shown in Figure 5-3A. Consider a region made up of several faces. If we needed to access the 50th face of a region object using a traditional BLOB storage mechanism, one would have to load and sequentially traverse the entire BLOB until the desired face would be found. Further, since the face objects can be of variable length, the location of the 50th face cannot be easily computed without extra support built in to the BLOB. In order to avoid an undesirable sequential traversal of the BLOB, we introduce the notion of offsets to describe structure. Each hierarchical level of a structure in a structure index stored in a BLOB is made up of two components (corresponding to the two components of the general structure index described above). The first component contains offsets that represent the location of specific sub-objects. The second component represents the sub-objects themselves. We define offsets to have a fixed size; thus, the location of the *i*th face can be directly determined by first calculating the location of the *i*th offset and then reading the offset to find the location of the face. Figure 5-5 shows a structured object with internal offsets.

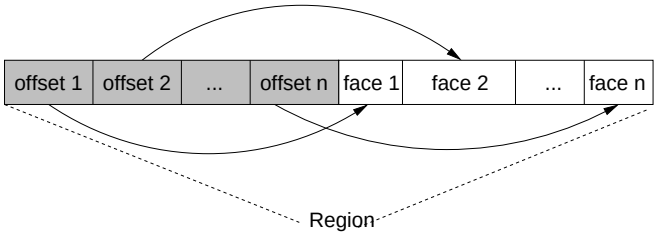


Figure 5-5. A structured object consisting of n sub-objects and n internal offsets

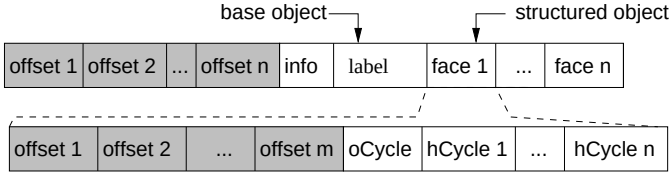


Figure 5-6. A structured object consisting of a base object and structured sub-objects

The recursive nature of hierarchical structures allows us to generalize the above description. Each sub-object can itself have a structure like the region described above. Objects at the same level are not required to have the same structure; thus, at any given level it is possible to find both structured sub-objects and base objects (raw data). For example, we can extend the structure of a region object so that it is made up of a collection of faces each of which contains an outer cycle and zero or more hole cycles, which in turn are made up of a collection of segments. Segments can be implemented as a pair of (x, y)-coordinate values. This example is illustrated in terms of structured and base objects in Figure 5-6 where the top level object represents a region with an information part, a label, and one of its face sub-objects.

In general, a specific structure index implementation must be defined with respect to the underlying unstructured storage medium. Because we have to use BLOBs as the only alternative in a database context, we are forced to embed both the structure index and the data into a BLOB. Thus, using an offset structure embedded within the data itself is an appropriate solution. However, this may not be ideal in all cases. For instance, one could implement a structure index for data stored in flat files. In this case, the structure index and the data could be represented in seperate files. In general, the

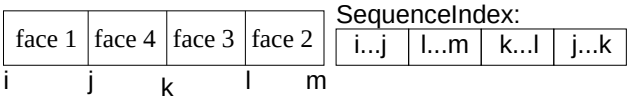


Figure 5-7. An out-of-order set of data blocks and their corresponding sequence index

structure index concept must be adapted to the capabilities of the user's desired storage medium for implementation.

- iBLOB sequence index: tracking data order for updates

Different DBMSs provide different implementations of the BLOB type with varied functionalities. However, most advanced BLOB implementations support three operations at the byte level, namely, random read and append (write bytes at end of BLOB), truncate (delete bytes at end) and overwrite (replace bytes with another block of bytes of the same or smaller length).

Structured large objects require the ability to update sub-objects within a structure. Specifically, they require random updates which include insertion, deletion and the ability to replace data with new data of arbitrary size. Examples are the replacement of a segment by several segments in a cycle of a region object, or adding a new face. Given a large region object, updating it entirely for each change in a face, cycle or segment becomes very costly when stored in BLOBs (update problem). Thus, it is desirable to update only the part of the structure that needs updating. For this purpose, we present a novel sequence index concept that is based on the random read and data append operations supported by BLOBs Extra capabilities provided by higher level BLOBs are a further improvement and serve for optimization purposes. The sequence index concept is based on the idea of physically storing new data at the end of a BLOB and providing an index that preserves the logically correct order of data.

Consequently, data will have internal fragmentation and will be physically stored out-of-order, as illustrated in Figure 5-7. In this figure, the data blocks (with start and end byte addresses represented by letters under each boundary) representing faces should

Figure 5-8. The initial in-order and defragmented data and sequence index.

be read in the order $1, 2, 3, 4$, even though physically they are stored out-of-order in the BLOB (we will study the possible reasons shortly). By using an ordered list of physical byte address ranges, the sequence index specifies the order in which the data should be read for sequential access. The sequence index from Figure 5-7 indicates that the block $[i \ldots j]$ must be read first, followed by the block $[l \ldots m]$, etc.

Based on the general description of the sequence index given above, we now show how to apply it as a solution to the update problem. Assume that the data for a given structured object is initially stored sequentially in a BLOB, as shown in Figure 5-8. Suppose further that the user then makes an insertion at position k in the middle of the object. Instead of shifting data after position k within the BLOB to make room for the new data, we append it to the BLOB as block $[j \ldots l]$, as shown in Figure 5-9. By modifying the sequence index to reflect the insertion, we are able to locate the new data at its logical position in the object.

Figure 5-10 illustrates the behavior of the sequence index when a block is intended to be deleted from the structured object. Even though there is no new data to append to the BLOB, the sequence index must be updated to reflect the new logical sequence. Because the BLOB does not actually allow for the deletion of data, the sequence index is modified in order to prevent access to the deleted block $[m \ldots n]$ of data. This can result in internal fragmentation of data in the iBLOB which can be managed using a special resequence operation shown later in the iBLOB interface.

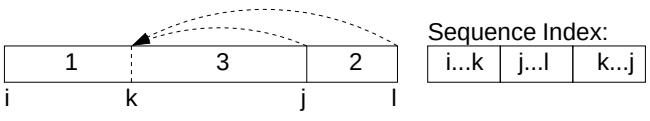


Figure 5-9. A sequence index after inserting block $[j \ldots l]$ at position k.

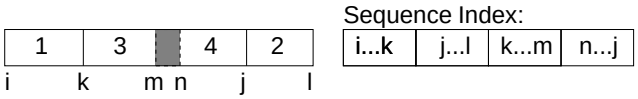


Figure 5-10. A sequence index after deleting block $[m \ldots n]$.

Finally, Figure 5-11 illustrates the case of an update where the values of a block of data $[o \ldots p]$ as a portion of block $[j \ldots l]$ are replaced with values from a new block $[l \ldots q]$. For this kind of update, it is possible for the new set of values to generate a block size different from that of the original block being replaced.

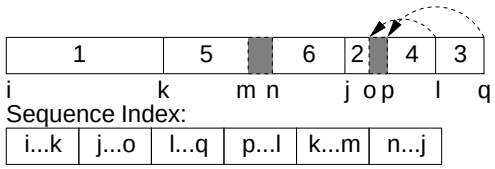


Figure 5-11. A sequence index after replacing block $[o \ldots p]$ by block $[l \ldots q]$.

iBLOBs enhance BLOBs by providing support for truncate and overwrite operations at the higher component level of an application object's structure. The truncate operation in BLOB (delete bytes at end) is enhanced in iBLOB with a remove function which can perform deletion of components at any location (beginning, middle or at the end of structure) as shown in Figure 5-10. The overwrite operation in BLOB (replace byte array with another of same length) is enhanced in iBLOB with a combination of remove and insert functions and sequence index adjustments, to perform the overwrite of components with other components of different sizes as shown in Figure 5-10.

- The iBLOB interface

In this section, we present a generic interface for constructing, retrieving and manipulating iBLOBs. Within this iBLOB interface, we assume the existence of the following data types: the primitive type Int for representing integers, Storage as a storage structure handle type (i.e., blob handle, file descriptor, etc.), Locator as a reference type for an iBLOB or any of its sub-objects, Stream as an output channel for reading byte blocks of arbitrary size from an iBLOB object or any of its sub-objects, and

$$create: \rightarrow iBLOB \quad (5\text{–}1)$$

$$create: Storage \rightarrow iBLOB \quad (5\text{–}2)$$

$$create: iBLOB \rightarrow iBLOB \quad (5\text{–}3)$$

$$copy: iBLOB \times iBLOB \rightarrow iBLOB \quad (5\text{–}4)$$

$$locateiBLOB: iBLOB \rightarrow Locator \quad (5\text{–}5)$$

$$locate: iBLOB \times Locator \times Int \rightarrow Locator \quad (5\text{–}6)$$

$$getStream: iBLOB \times Locator \rightarrow Stream \quad (5\text{–}7)$$

$$insert: iBLOB \times data \times Int \times Locator \times Int \rightarrow iBLOB \quad (5\text{–}8)$$

$$insert: iBLOB \times iBLOB \times Locator \times Int \rightarrow iBLOB \quad (5\text{–}9)$$

$$remove: iBLOB \times Locator \times Int \rightarrow iBLOB \quad (5\text{–}10)$$

$$append: iBLOB \times data \times Int \times Locator \rightarrow iBLOB \quad (5\text{–}11)$$

$$append: iBLOB \times iBLOB \times Locator \rightarrow iBLOB \quad (5\text{–}12)$$

$$length: iBLOB \times Locator \rightarrow Int \quad (5\text{–}13)$$

$$count: iBLOB \times Locator \rightarrow Int \quad (5\text{–}14)$$

$$resequence: iBLOB \rightarrow iBLOB \quad (5\text{–}15)$$

Figure 5-12. The standardized iBLOB interface

data as a representation of a base object. Figure 5-12 lists the operations offered by the interface. We use the term *l-referenced object* to indicate the object that is referred to by a given locator *l*. The following descriptions for these operations are organized by their functionality:

- **Construction and Duplication**: An iBLOB object can be constructed in three different ways. The first constructor $create()$ (5–1) creates an empty iBLOB object. The second constructor $create(sh)$ (5–2) constructs an iBLOB object from a specific storage structure handle sh such as a BLOB object handle or a file descriptor. The third constructor $create(s)$ (5–3) is a copy constructor and builds a new iBLOB object from an existing iBLOB object s. Similarly, an iBLOB object s_2 can also be copied into another iBLOB object s_1 by using the $copy(s_1, s_2)$ operator (5–4).

- **Internal Reference**: In order to provide access to an internal sub-object of an iBLOB object, we need a way to obtain the reference of such a sub-object. The sub-object referencing process must start from the topmost hierarchical level of the iBLOB object s whose locator l is provided by the operator $locateiBLOB(s)$ (5–5). From this locator l, a next level sub-object can be referenced by its slot i in the operator $locate(s, l, i)$ (5–6).

- **Read and Write**: Since iBLOBs support large objects which may not fit into main memory, we provide a stream based mechanism through the operator $getStream(s, l)$ (5–7) to consecutively read arbitrary size data from any l-referenced object. The stream obtained from this operator behaves similarly to a common file output stream. Other than reading data, the interface allows insertion of either a base object d of specified size z through the operator $insert(s, d, z, l, i)$ (5–8) or an entire iBLOB object s_1 through the operator $insert(s, s_1, l, i)$ (5–9) into any l-referenced object at a specified slot i. A base object d such as in the operator $append(s, d, z, l)$ (5–11) or a iBLOB object s_1 such as in operator $append(s, s_1, l)$ (5–12) can be appended to an l-referenced object. This is effectively the same as inserting the input as the last sub-object of the referenced object. The operator $remove(s, l, i)$ (5–10) removes the sub-object at slot i from the parent component with Locator l.

- **Properties and Maintenance**: The actual size of an l-referenced object is obtained by using the operator $length(s, l)$ (5–13) while the number of sub-objects of the object is provided by the operator $count(s, l)$ (5–14). Finally, the operator $resequence(s)$ (5–15) reorganizes and defragments the iBLOB object s collapsing its sequence index such that it contains a single range. This operation effectively synchronizes the physical and logical representations of the iBLOB object and minimizes the storage space.

To test the functionality we have implemented the iBLOB data type in Oracle using object oriented extensions and programming API of the DBMS. Due to space constraints, we have omitted the iBLOB implementation details in this article.

Each operator in the TSS interface can be implemented using the corresponding iBLOB interface operator. For e.g., to implement $get(region.face[1].outerCycle.segment[1])$, we first use $locateiBLOB$ (5–5) to get a Locator to the iBLOB, then use $locate()$ (5–5) repeatedly to move across levels and navigate to the required component (i.e., first segment), and finally, $getStream()$ (5–7) to retrieve the first segment of the outerCycle in fifth face. Other TSS interface functions like set, $baseObjectCount$ and $subObjectCount$ can also be implemented in a similar manner.

we provide a novel solution to store and manage complex application objects (i.e., variable length, structured, hierarchical data) by introducing a new mechanism for handling structured objects inside DBMSs. The combination of type structure

specification and iBLOBs provides the necessary tools to easily implement type systems in databases.

5.1.4 Evaluation of the iBLOB approach

The iBLOB approach is the first attempt to solve the problem of storing large structured objects in a database context using the object oriented paradigm. By redesigning the way of storing data in BLOBs, we have extended a very low level binary storage to a much smarter and higher level storage that is more suitable for applications with large structured objects. To demonstrate that we do not only save the type system implementer from tedious serialization and deserialization tasks, but also provide them with more efficient access and update operations for data stored in iBLOBs, we evaluation our iBLOB from both the storage and performance perspective.

We have implemented our iBLOB data type in Oracle11g based on the standard data type (SDT) BLOB. We select the US state maps from US Census Bureau(http://www.census.gov/) as our testing subjects. We have downloaded 56 maps that are at the state layer from the 2010 TIGER/Line Shapefiles. The maps are stored in the format of *shapfiles*(*.shp*), which is a very popular file format for storing 2D shapes, and is used in a large number of well known applications such as the ArcGIS from ESRI and the TIGER Shapefiles from US Census Bureau. Shapefiles spatially describe geometries: points, polylines, and polygons. These, for example, could represent water wells, rivers, and lakes, respectively. Unlike other shape representation files like the KML files supported by Google maps which are human readable, the content of a shapefile is in a binray form thus is not visible to users, and must be accessed from program libraries or softwares like ArcGIS. The detailed specification of shapefiles is available in [35]. Since we use the dataset from the TIGER Shapefiles as our testing data, which contains only 2D region objects, we only briefly describe how regions are stored in shapefiles in this section.

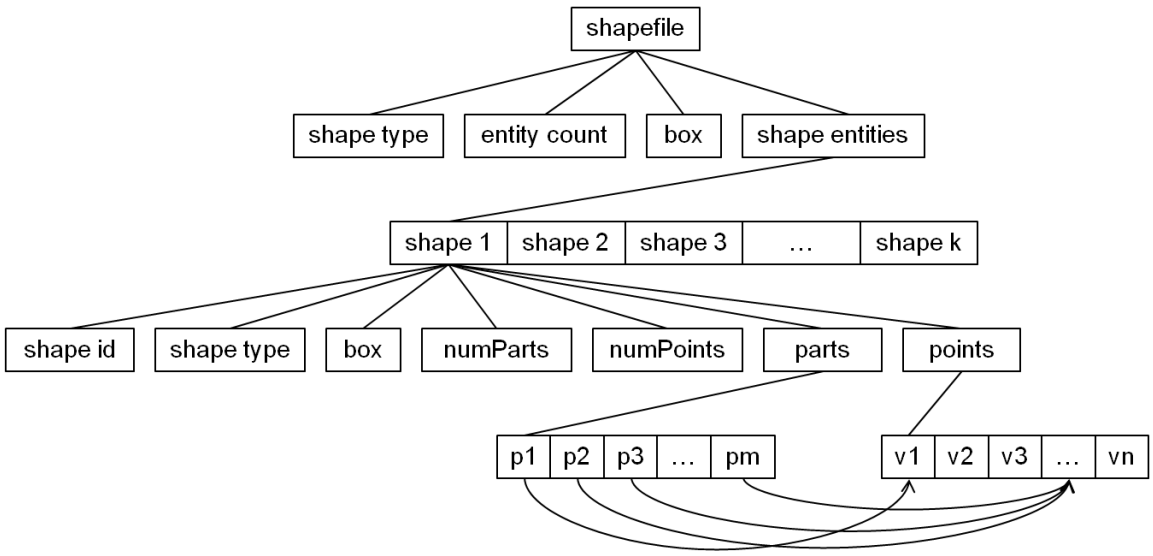


Figure 5-13. The structured representation of a polygon type in shapefiles.

A polygon shape in a shapefile has the attributes of the shape type, the entity count, the bounding box, and the entity list. A shape type is a integer value that indicates the type(e.g. 5 for polygon). The entity count indicates the number of entities included in this shapefile where entities are disjoint independent polygon shapes. For example, a country can have multiple islands where each island is a polygon entity. The bounding box of a shapefile is a pair of points that can be used to compute the minimum bounding box of a shape. Finally, a shapefile consists of a list of entities which have further structures inside. Each entity is a polygon that consists of one or more rings. A ring is a connected sequence of four or more points that form a closed, non-self-intersecting loop. A polygon may contain multiple outer rings. The order of vertices or orientation for a ring indicates which side of the ring is the interior of the polygon. The neighborhood to the right of an observer walking along the ring in vertex order is the neighborhood inside the polygon. Vertices of rings defining holes in polygons are in a counterclockwise direction. Vertices for a single, ringed polygon are, therefore, always in clockwise order. The rings of a polygon are referred to as its parts. Figure 5-13 presents a hierarchical tree representation that describes a polygon shapefile.

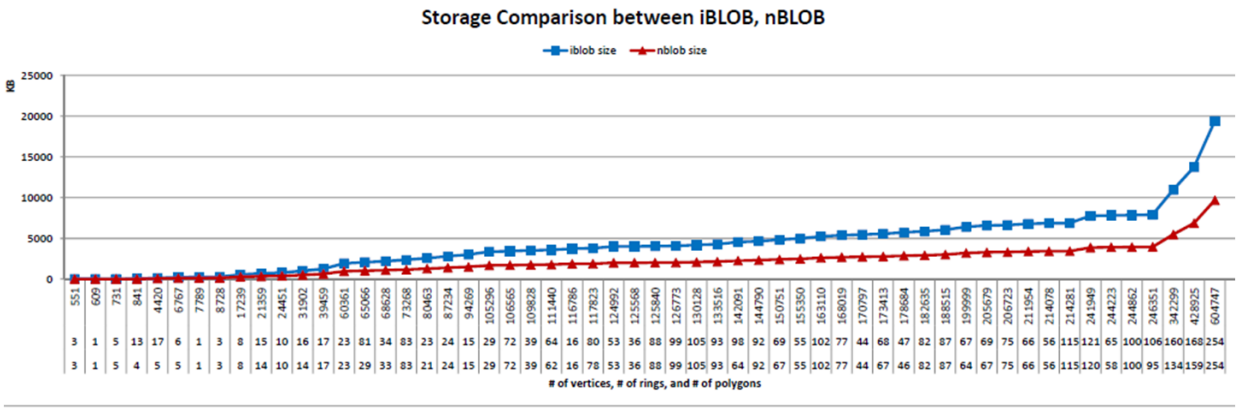


Figure 5-14. The storage comparison between iBLOB and nBLOB for 56 TIGER shapefiles.

In order to evaluate the performance of the iBLOB under real datasets with different sizes, we parse the TIGER shapefiles and store the polygon shapes in our iBLOBs. Since the original TIGER shapefiles range from 9KB to 9MB, we have a good diversity in terms of storage size for iBLOBs which allows us to check the performance of iBLOBs under different scales. Further, to compare with iBLOBs, we evaluate another approach which is the only alternative option for storing objects like shapefiles in a database context. We evaluate the traditional BLOB approach. A straight forward and simple solution for storing large objects in databases like Oracle is to store the files directly in BLOBs. This allows SQL queries to load the entire file from a BLOB and parse it in main memory, which requires a serialization for writing data to BLOBs and a deserialization for reading data from BLOBs. We have implemented this strategy and have tested it based on the same datasets. Let us call the BLOB approach the *nBLOB* approach to distinguish from our iBLOB approach. We compare the two approaches from four perspectives, storage, random reads, random inserts, and random deletes.

Figure 5-14 compares the performance of the three approaches from the storage perspective. We can observe that the storage of all three approaches linearly increase. The storage cost of an iBLOB is higher than the nBLOB approach and is lower than the XMLType approach. This is reasonable because the iBLOB introduces overhead when

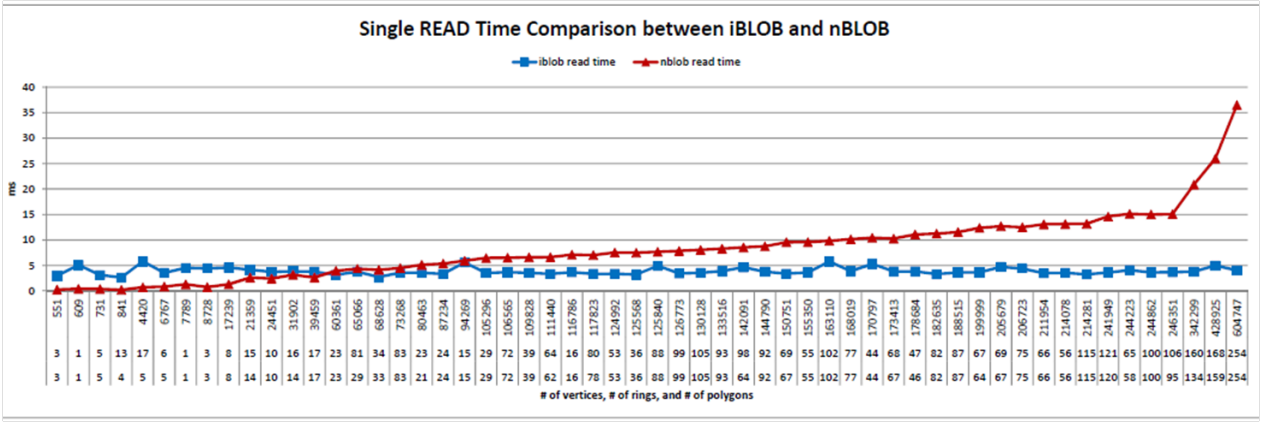


Figure 5-15. The random single read time comparison between iBLOB and nBLOB for 56 TIGER shapefiles.

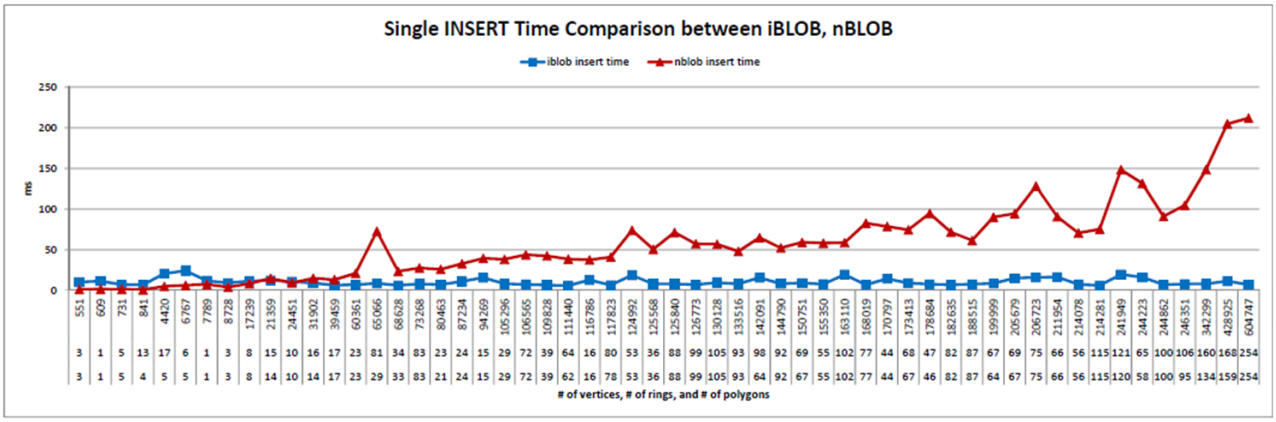


Figure 5-16. The random single insert time comparison between iBLOB and nBLOB for 56 TIGER shapefiles.

preserving the structure of objects inside. On the other hand, since iBLOB maintains structure in a binary fashion which is not readable to human being, thus it has lower cost than the XMLType, which reveals the internal structure by describing the structure using text.

Figure 5-15 shows the comparison between iBLOB and nBLOB approaches for the random single read operation. The random single read operation first generates a list of random numbers that are used to access a random vertex of a random entity in the polygon shape. Then the vertex is read and printed. It shows two different trends with respect to the increase of storage size for the read operation in Figure 5-15. The

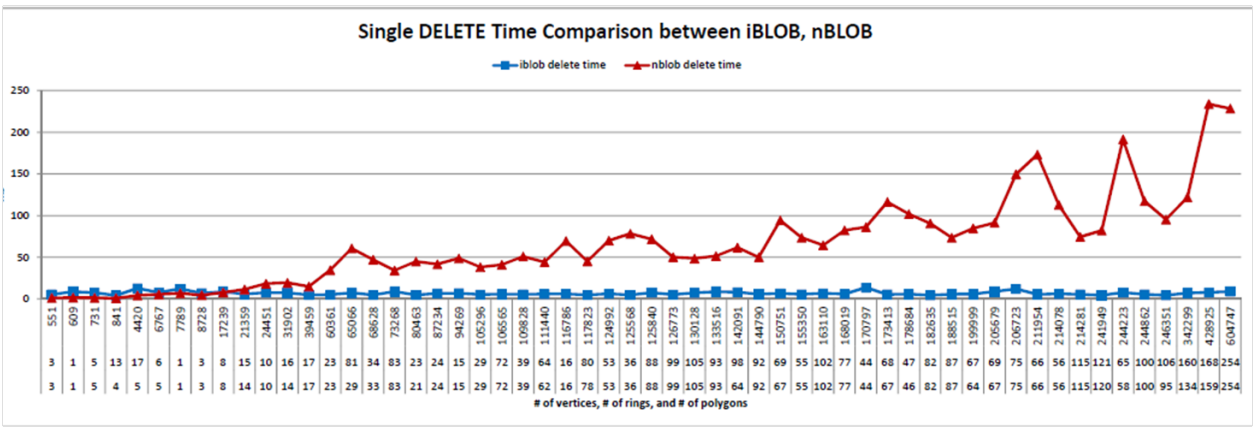


Figure 5-17. The random single delete time comparison between iBLOB and nBLOB for 56 TIGER shapefiles.

time consumed by the single read operation in the nBLOB approach increase linearly while the read on the iBLOB stays almost constant. The reason for this is that the only facts that may effect the performance of read operations in an iBLOB are the number of reads performed and the size of each read operation. On the other hand, the nBLOB approach only provides a storage for shapes, and every read requires the deserialization of the entire object to main memory first. However, when the size of the object stored is relatively small, the deserialization has less impact on the read operation, thus the nBLOB approach maybe superior than the iBLOB approach. It is because the iBLOB approach will issue more than one I/O reads in order to get to the particular vertex, while nBLOB only needs to read once. When object size is not big, then the number of I/O dominants the read operation. Similarly, we observe the same trend in the random single insert operation and random single delete operation for both approaches in Figure 5-16 and Figure 5-17.

Therefore, we can conclude from the comparisons between the two approaches that the operations like read, insert and delete on iBLOBs are much more scalable than the ones one the simple BLOB approach.

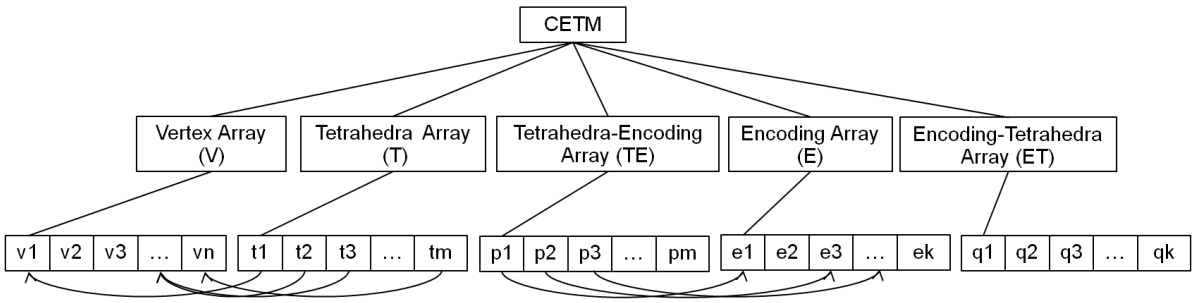


Figure 5-18. The tree representation of CETM.

5.1.5 Integration of the CETM in Databases Based on iBLOBs

We have proposed a solution for representing 3D volumes with a compact tetrhadralization based data structure in Section 4.2. We show in this section how the CETM data structure can be integrated into a database context based on our iBLOBs. There exists only two options for storing the The CETM data structure in databases. Either storing it using the raw BLOB approach or applying our newly designed iBLOB approach. In this section, we show the results of applying both approaches and compare the performance on both.

Storing CETM data structure to an iBLOB is rather straight forward. We first map the CETM data structure into a hierarchical tree structure as described in Figure 5-18, then we create a program to insert the internal nodes of the tree as the structured objects in an iBLOB, and the leaf nodes as the base objects in an iBLOB, by using the insert API from the iBLOB interface. Although storing data to iBLOBs introduces addtional storage overhead, they are all handled by the iBLOBs and do not require additional serialization and deserialization procedures.

When storing CETM data structure to a raw BLOB storage, we need to first carefully design the serialization and deserialization procedures to convert the in-memory data structures to sequential binary representations. Additional storage is also introduced (although not as much as the iBLOB approach) in the serialization procedure. The access operations like read and write need to be implemented to retrieve data from the

Table 5-1. The storage cost and the topological relation retrieval performance of CETM data structure implemented based on the iBLOB and the BLOB.

	$m500$	$m471$	$m58$	$m531$	$m67$
CETM size (byte)	48,925	321,072	367,466	2,332,140	20,101,923
iBlob size (byte)	51,086	334,372	383,912	2,433,892	21,025,766
Blob size (byte)	51,086	334,372	383,912	2,433,892	21,025,766
R03 (100, 3)					
Tetetrahedra Visited	11	12	26	14	9
iBlob Access Time (ms)	33.672	38.12	254.089	120.518	41.435
Blob Access Time (ms)	9.367	13.19	12.776	40.603	123.913
R13 (100, 2, 3)					
Tetetrahedra Visited	2	5	6	3	6
iBlob Access Time (ms)	20.124	49.432	89.383	26.906	40.123
Blob Access Time (ms)	8.336	11.285	13.064	32.455	109.225

BLOB storage, which usually requires deserializing the entire object from BLOB to a main memory data structure and read or write.

Table 5-1 shows a comparison between the two approaches for storing CETM data structure in databases. We can observe that the BLOB based approach has a better performance over small size data that is than 3MB. But when the data scale exceeds 5MB, the iBLOB approach out performs the BLOB approach. This matches what we have observed in the experimental study of the iBLOBs in Section 5.1.4, that the iBLOB approach is more stable and scalable than the raw BLOB approach.

5.2 Integration of 3D Spatial Operations and Predicates in SQL

5.2.1 Integration of the Topological Predicates in SQL Based on the NCM Model

Using the Neighborhood Configuration Model (NCM) with its topological encodings described in Section 3.2.1.2, we can identify the topological relationships between any two given complex volume objects in the form of a 15 bit binary encoding. However, to integrate topological relationships into spatial databases as selection and join conditions in spatial queries, topological predicates with the signature $volume \times volume \rightarrow bool$ are needed and have to be formally defined. For example, a query like “Find all buildings that meet the Physics building” requires a topological predicates like meet as a selection condition of a spatial join. Assuming a relation buildings with attributes bname of type

string and shape of type volume, we can express the query in a SQL-like style as follows:

```
SELECT b2.pname FROM buildings b1, buildings b2
WHERE  b1.bname='Physics' and b1.shape meet b2.shape
```

The 15 bit topological encodings, which represent topological relationships between volumes, describes a total of 697 topological relationships between two complex volumes and a total of 72 topological relationships between two simple volumes. Therefore, a maximum of 697 topological predicates can be defined to provide an exclusive and complete coverage of all possible topological relationships. However, users will not be interested in such a large, overwhelming, and predefined collection of detailed predicates since they will find it difficult to distinguish, remember, and handle them [25]. Instead, a reduced and manageable set of predicates will be preferred. Such a set should be user-defined and/or application specific since different applications may have different criteria for the distinction of directional relationships. For example, one application could require a clear distinction between the topological relationships *only_meet_on_face*, *only_meet_at_point*, *only_overlap*, *exist_overlap_and_meet_on_face* whereas another application could abandon a distinction between the four predicates and regard them all as overlap. Thus, flexibility is needed to enable users to define their own set of topological predicates and select their own names for these predicates. A clustering strategy has been introduced in [88] to use topological predicate clusters to produce a reduced set of topological predicates for two 2D spatial objects. In this approach, a set of constraint rules called clustering rules is given to define the reduced set of predicates which are mutually exclusive and cover all the predicates enumerated from the 9-intersection matrix. In this paper, in order to offer the user more flexibility of defining topological predicates, we take a different approach. We provide two levels of topological predicates. Level 1 comprises so-called existential topological predicates, and Level 2 contains so-called derived directional predicates. The predicates of the

higher level are derived and built from the predicates of the lower level. All predicates ease the task of formulating complex topological queries.

In our NCM model, we identify the topological relationship between two given complex volume objects by checking the existence of the 15 neighborhood configurations. Each of the 15 neighborhood configurations characterizes one unique topological interaction between two volume objects. Hence, instead of defining 697 mutually exclusive topological predicates to cover all 697 possible topological relationships, we define existential topological predicates for identifying the existence of the topological features.

Level 1 comprises 14 predefined existential topological predicates that we specify in Definition 48 and that ensure the existence of a particular basic topological interaction between at least one part of a volume object A and at least one part of a volume object B. Each existential topological predicates corresponds to the value of one bit in the encoding. The reason that we only need to define 14 existential topological predicates is that, according to Lemma 1, the fourth bit of a topological encodings is always 1. These predicates provide an interface for users to define their own derived topological predicates at Level 2. Therefore, we assume that the implementation of the 15 existential topological predicates is provided in a spatial type system (spatial algebra, spatial extension package) and that these predicates can be embedded into spatial SQL queries.

Definition 48. *Let A, B be two complex volumes and let $NCM(A, B)$ denote the function that computes the topological encoding E ($E = NCM(A, B)$) between volume A and B. Please note that the function NCM is an asymmetric function, so that $NCM(A, B) \neq NCM(B, A)$. Then the existential topological predicates are defined as:*

A share_interior_with B $\Leftrightarrow E_1 = 1 \wedge E = NCM(A, B)$

A has_interior_outside B $\Leftrightarrow E_2 = 1 \wedge E = NCM(A, B)$

A does_not_contain_all_interior_of B $\Leftrightarrow E_3 = 1 \wedge E = NCM(A, B)$

A contains_boundary_of B $\Leftrightarrow E_5 = 1 \wedge E = NCM(A, B)$

A has_boundary_inside B $\Leftrightarrow E_6 = 1 \wedge E = NCM(A, B)$

A meets_from_inside B $\Leftrightarrow E_7 = 1 \wedge E = NCM(A, B)$

A meets_from_outside B $\Leftrightarrow E_8 = 1 \wedge E = NCM(A, B)$

A has_boundary_outside B $\Leftrightarrow E_9 = 1 \wedge E = NCM(A, B)$

A does_not_contain_all_boundary_of B $\Leftrightarrow E_{10} = 1 \wedge E = NCM(A, B)$

A contact_from_outside B $\Leftrightarrow E_{11} = 1 \wedge E = NCM(A, B)$

A contact_from_inside B $\Leftrightarrow E_{12} = 1 \wedge E = NCM(A, B)$

A is_contacted_from_inside_by B $\Leftrightarrow E_{13} = 1 \wedge E = NCM(A, B)$

A secretly_contacts B $\Leftrightarrow E_{14} = 1 \wedge E = NCM(A, B)$

A crosses B $\Leftrightarrow E_{15} = 1 \wedge E = NCM(A, B)$

For example, A *share_interior_with* B returns $true$ if a part of the interior of A is shared with a part of the interior of B; this does not exclude the existence of other basic topological predicates between other parts of A and B, e.g., A *has_interior_outside* $B = true$. Hence, we have an existential viewpoint. The predicate A *secretly_contacts* B describes a scenario where the contacting point between A and B is not visible to the outside. It is trivial to prove that by using the existential topological predicates and the logical operators $\neg$, $\vee$, and $\wedge$, we can obtain a complete coverage and mutual distinction of all possible 697 topological relationships from the NCM model. This will enable users to define any set of composite topological predicates for their own needs and applications. For example, the topological relationship between A and B that is identified by the topological encoding 111111011110011 (see Figure 3-26) can be expressed by the following unique Boolean expression:

A *share_interior_with* $B \wedge A$ *has_interior_outside* $B \wedge$

A *does_not_contain_all_interior_of* $B \wedge A$ *contains_boundary_of* $B \wedge$

A *has_boundary_inside* $B \wedge \neg(A$ *meets_from_inside* $B) \wedge$

A *meets_from_outside* $B \wedge A$ *has_boundary_outside* $B \wedge$

A *does_not_contain_all_boundary_of* $B \wedge A$ *contact_from_outside* $B \wedge$

$\neg(A$ *contact_from_inside* $B) \wedge \neg(A$ *is_contacted_from_inside_by* $B) \wedge$

A *secretly_contacts* $B \wedge A$ *crosses* B

Based on the nine existential predicates from Level 1, users can specify their own derived topological predicates at Level 2. Derived topological predicates are composite topological predicates that are built using one or more of the existential topological predicates and other already defined derived topological predicates. As examples, we construct the traditional 8 topological predicates that are mutually exclusive and covers all 697 topological relationships. The 8 topological predicates are disjoint, meets, inside, contains, coveredby, covers, equals and overlaps. These predicates are consistent with the ones defined in the 2D space [88]. Definition 49 shows the specification of these predicates.

Definition 49. *Let* A, B *be two complex volumes, then the topological predicates disjoint, meets, inside, contains, coveredby, covers, equals and overlaps are defined as:*

$$
\begin{aligned}
A disjoint B \quad & \Leftrightarrow \neg(A \text{ share\_interior\_with } B) \wedge \neg(A \text{ meets\_from\_outside } B) \\
& \wedge A \text{ has\_boundary\_outside } B \wedge \neg(A \text{ contact\_from\_outside } B) \\
& \wedge A \text{ does\_not\_contain\_all\_boundary\_of } B \\
A \text{ meets } B \quad & \Leftrightarrow \neg(A \text{ share\_interior\_with } B) \wedge \\
& (A \text{ meets\_from\_outside } B \vee A \text{ contact\_from\_outside } B) \\
A \text{ inside } B \quad & \Leftrightarrow \neg(A \text{ has\_interior\_outside } B) \wedge A \text{ has\_boundary\_inside } B \\
& \wedge \neg(A \text{ meets\_from\_inside } B) \wedge \neg(A \text{ contact\_from\_inside } B) \\
& \wedge A \text{ does\_not\_contain\_all\_boundary\_of } B \\
A \text{ contains } B \quad & \Leftrightarrow \neg(A \text{ does\_not\_contain\_all\_interior\_of } B) \wedge \\
& A \text{ contains\_boundary\_of } B \wedge \neg(A \text{ meets\_from\_inside } B) \wedge \\
& A \text{ has\_boundary\_outside } B \wedge \\
& \neg(A \text{ is\_contacted\_from\_inside\_by } B) \\
A \text{ coveredby } B \quad & \Leftrightarrow \neg(A \text{ has\_interior\_outside } B) \wedge (((A \text{ has\_boundary\_inside } B \\
& \vee A \text{ does\_not\_contain\_all\_boundary\_of } B) \wedge \\
& A \text{ meets\_from\_inside } B) \vee A \text{ contact\_from\_inside } B) \\
A \text{ covers } B \quad & \Leftrightarrow \neg(A \text{ does\_not\_contain\_all\_interior\_of } B) \wedge \\
& (((A \text{ contains\_boundary\_of } B \vee A \text{ has\_boundary\_outside } B) \wedge \\
& A \text{ meets\_from\_inside } B) \vee A \text{ is\_contacted\_from\_inside\_by } B) \\
A \text{ equals } B \quad & \Leftrightarrow \neg(A \text{ has\_interior\_outside } B) \wedge A \text{ meets\_from\_inside } B \\
& \wedge \neg(A \text{ does\_not\_contain\_all\_interior\_of } B)
\end{aligned}
$$

Table 5-2. The number of topological relationships assigned to each of the 8 traditional topological predicates.

disjoint	*meets*	*inside*	*contains*	*coveredby*	*covers*	*equals*	*overlaps*
1	7	1	1	7	7	1	672

A overlaps $B \Leftrightarrow A$ crosses $B \vee A$ secretly_contacts $B \vee$

$((A$ contains_boundary_of $B \vee$

A is_contacted_from_inside_by $B)$

$\wedge (A$ has_boundary_inside $B \vee A$ meets_from_outside $B \vee$

A contact_from_inside $B)) \vee$

A does_not_contain_all_boundary_of B

$\vee ((A$ meets_from_outside $B \vee A$ contact_from_outside $B) \wedge$

$(A$ has_boundary_inside $B \vee A$ meets_from_inside $B \vee$

A contact_from_inside $B)) \vee (A$ has_boundary_outside $B \wedge$

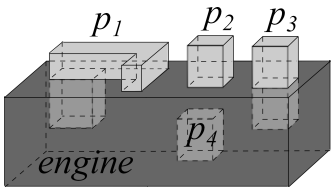
$(A$ has_boundary_inside $B \vee A$ contact_from_inside $B \vee$

$(A$ meets_from_inside $B \wedge$

A does_not_contain_all_boundary_of $B)))$

In order to prove the mutual exclusion of the obtained topological predicates as well as the complete coverage of all basic topological relationships by them, we have developed a program to check all 697 topological encodings. We mark each encoding with the names of the corresponding predicates that yield true for the encoding. As a result, we observe that all encodings are marked with only one predicate name (mutual exclusion), and there does not exist any encoding that is not marked (complete coverage). Table 5-2 shows the number of encodings that makes each corresponding predicate true.

As we have seen, all derived topological predicates are defined on the basis of the existential topological predicates given in Definition 48. We have proposed the derived topological predicates here only as examples. It is up to the users to define their own predicates and to select their own predicate names according to their needs



(a)

Q1 result: p_2, p_4
Q2 result: p_1, p_2, p_4
Q3 result: p_1

(b)

Figure 5-19. An engine and several parts (a) and the results of the queries Q1, Q2, and Q3 (b).

and their applications. But for this purpose, they need data definition language (DDL) suport in SQL in order to be able to specify their own topological predicates and then to use them in SQL queries. We propose a new `CREATE TPRED` clause that enables users to syntactically specify derived topological predicates in an easy manner. For creating Boolean expressions we provide the logical operators & for $\wedge$, | for $\vee$, and ~ for $\neg$. Alternatively, we could use the logical operators `AND`, `OR`, and `NOT` built into SQL, but this could also lead to a confusion with their normal usage. Since all directional predicates are binary predicates, we have omitted their arguments for reasons of simplicity and better readability. As example we show the SQL specification of the topological predicates meets and equals.

```
CREATE TPRED meets
  (~share_interior_with & ( meets_from_outside | contact_from_outside ))


CREATE TPRED equals
  (~has_interior_outside & meets_from_inside &
  ~does_not_contain_all_interior_of)
```

The use of the topological predicates in SQL queries is quite standard and therefore familiar to the user. However, the semantics that can be expressed by these predicates can be quite different and powerful. For example, assuming the following two relations:

```
parts (pname:string, body:volume);
machines (mid:string, type:string, body:volume);
```

We can pose the following queries:

Q1: Determine the parts that attach but not penetrate the engine body.

```
SELECT p.pname FROM parts p, machine m
WHERE  m.type='engine' and p.body meets m.body
```

Q2: Determine the parts that are attached to the surface of the engine body.

```
SELECT p.pname FROM parts p, machine m
WHERE  m.type='engine' and ( p.body meets_from_inside m.body or
       p.body meets_from_outside m.body )
```

Q3: Determine the parts that both attach to the surface of the engine body from outside and penetrate the engine body.

```
SELECT p.pname FROM parts p, machine m
WHERE  m.type='engine' and p.body meets_from_outside m.body and
      p.body overlaps m.body
```

The above three queries return different results, which are illustrated in Figure 5-19.

5.2.2 Detour: Integration of the Spatio-temporal Directional Predicates in SQL

In this section, we discuss how cardinal direction developments can be integrated into spatio-temporal databases and query languages. This requires the formal definition of cardinal direction developments as binary predicates since it will make the query processing easier when using pre-defined predicates as selection conditions. In the following part, we define some important predicates which will be sufficient for most queries on cardinal direction developments between moving objects.

First of all, we give the definition of existential direction predicates. This type of predicates finds out whether a specific cardinal direction existed during the evolution of moving objects. For example, a query like "Find all ships that appeared north of ship Fantasy" belongs to this category. It requires a predicate named *exists_north* as a selection condition of a join. This predicate can be defined as follows,

Definition 50. *Given two moving points* $A, B \in MPoints$*, their cardinal direction development* $DEV(A, B) = d_1 \triangleright d_2 \triangleright \ldots \triangleright d_n$ *with* $n \in \mathbb{N}$ *and* $d_i \in CD$ *or* $d_i = \perp$ *for all* $1 \leq i \leq n$*. Then we define the existential direction predicate* exists_north *as*

$$exists\_north(A, B) = true \overset{\text{def}}{\Leftrightarrow} \exists 1 \leq i \leq n : d_i = N$$

Definition 50 indicates that the predicate *exists_north* is true if the direction *north* exists in the sequence of the cardinal direction development. It can help us define the above query. Assume that we have the following relation schema for ships

```
ships(id:integer, name:string, route:mpoint)
```

The query can be expressed using an SQL-like query language as follows:

```
SELECT s1.name FROM ships s1, ships s2
WHERE  s2.name = 'Fantasy' AND exists_north(s1.route, s2.route);
```

The other existential cardinal direction predicates *exists_south*, *exists_east*, *exists_west*, *exists_sameposition*, *exists_northeast*, *exists_southeast*, *exists_northwest*, and *exists_southwest* are defined in a similar way.

Another important category of predicates expresses that one moving object keeps the same direction with respect to another moving object. For example, assume that there is a group of ships traveling from north to south and each ship follows the ship in front of the group. Now the leader of the group wants to know which ships belong to the group. The problem is to find out which ships are keeping a northern position with respect to the leading ship.

Definition 51. *Given two moving points* $A, B \in MPoints$*. The predicate* keeps_north *is defined as*

$$\begin{aligned} keeps\_north(A, B) \; = \; & exists\_north(A, B) && \wedge \neg exists\_south(A, B) \\ & \wedge \neg exists\_southeast(A, B) && \wedge \neg exists\_east(A, B) \\ & \wedge \neg exists\_sameposition(A, B) && \wedge \neg exists\_northwest(A, B) \\ & \wedge \neg exists\_northeast(A, B) && \wedge \neg exists\_southwest(A, B) \\ & \wedge \neg exists\_west(A, B) \end{aligned}$$

Definition 51 shows that the relationship *keeps_north* between two moving objects implies that the only existential direction predicate in the cardinal direction development of these moving objects is *exists_north* without any other existential direction predicates. In other words, we have $DEV(A, B) = N$.

We consider the above example and assume that the identifier of the leader ship is 1001. Then the query "Find all ships keeping a position north of the leader ship 1001" can be expressed as

```
SELECT s1.id FROM ships s1, ships s2
WHERE  s2.id = '1001' AND keeps_north(s1.route, s2.route);
```

The other predicates that express that one moving object remains in the same direction with respect to another moving object are *keeps_south*, *keeps_east*, *keeps_west*, *keeps_sameposition*, *keeps_northeast*, *keeps_southeast*, *keeps_northwest*, and *keeps_southwest*.

Another useful predicate checks for the transition between two cardinal directions in a cardinal direction development. The transition can be either a direct change or an indirect change through a set of intermediate directions. We name this predicate as $from\_to$. For example, the query "Find all ships that have traveled from the south to the north of the ship Fantasy" can be answered by using this predicate.

Definition 52. *Given two moving points* $A, B \in MPoints$*, their cardinal direction development* $\mathrm{DEV}(A, B) = d_1 \triangleright d_2 \triangleright \ldots \triangleright d_n$ *such that* $d_i \in CD$ *or* $d_i = \perp$ *for all* $1 \leq i \leq n$*, and two cardinal directions* $d', d'\prime \in \mathrm{CD}$*. We define the predicate* from_to *as follows:*

$$from\_to(A, B, d', d'\prime) = true \overset{\text{def}}{\Leftrightarrow} \quad d' \neq \perp \ \wedge \ d'\prime \neq \perp \wedge \\ \exists 1 \leq i < j \leq n : d_i = d' \ \wedge \ d_j = d'\prime$$

We formulate the above query as follows:

```
SELECT s1.id FROM ships s1, ships s2
WHERE  s2.name = 'Fantasy' AND
       from_to(s1.route, s2.route, 'S', 'N');
```

Finally, we define the predicate *cross_north* which checks whether a moving point traverses a large extent of the region in the north of another moving point.

Definition 53. *Given two moving points* $A, B \in MPoints$ *and their cardinal direction development* $\mathrm{DEV}(A, B) = d_1 \triangleright d_2 \triangleright \ldots \triangleright d_n$ *such that* $d_i \in CD$ *or* $d_i = \perp$ *for all* $1 \leq i \leq n$. *We define the predicate* crosses_north *as follows:*

$$\begin{aligned} cross\_north(A, B) = true \overset{\text{def}}{\Leftrightarrow} \quad & n \geq 3 \ \wedge \ \exists 2 \leq i \leq n-1 : \\ & (d_{i-1} = \mathrm{NW} \ \wedge \ d_i = N \ \wedge \ d_{i+1} = \mathrm{NE}) \vee \\ & (d_{i-1} = \mathrm{NE} \ \wedge \ d_i = N \ \wedge \ d_{i+1} = \mathrm{NW}) \end{aligned}$$

The query “Find all the ships that have crossed the north of ship Fantasy” can be expressed as follows:

```
SELECT s1.id FROM ships s1, ships s2
WHERE  s2.name = 'Fantasy' AND crosses_north(s1.route, s2.route);
```

The other predicates *cross_south*, *cross_east*, and *cross_west* can be defined in a similar way.

In this section, we have presented a three-phase solution for computing the cardinal directions between two moving points from an algorithmic perspective. We show the mapping of cardinal direction developments between moving points into spatio-temporal directional predicates and the integration of these predicates into the spatio-temporal query language of a moving objects database.

CHAPTER 6
CONCLUSIONS

The purpose of this research is to build and integrate a comprehensive component in extensible database management systems, which consists of a type system for complex three-dimensional data like surfaces and volumes, and query language extensions with most important operations and predicates like the intersection operation, the topological predicates, and the directional predicates for querying three-dimensional data. To achieve this goal, we have developed solutions at three main *levels of abstraction*.

We started with an *abstract level*, where we proposed the rigorous, mathematical definition of a comprehensive type system (algebra) for 3D spatial data including volumes, surfaces, and 3D lines. The concept was first proposed in the published work [89], and it is described in details and largely extended in this article. We also explore two most important qualitative spatial relationships, which are topological relationships and cardinal direction relationships, between two 3D spatial objects. We first designed a novel model called the *objects interaction matrix* (*OIM*) for modeling cardinal directions between two complex regions, which is first proposed in [100] and later extended in [66]. Then we extended the concept to the 3D space, and proposed the *objects interaction cube matrix* for 3D complex volume objects [23]. Further, we took a detour and considered the cardinal directions between two moving points which are special 3D points with z-dimension as time. We have developed the model to represent the development of the cardinal direction relationships between two moving points [21] and the algorithm for computing such a development [22] .

To explore the topological relationships between two spatial objects in the 3D space, we developed the neighborhood configuration model (*NCM*), which is based on the point set theory and point set topology. The foundation of this model is proposed in [99] and is largely extended and formally validated in this article. We have evaluated

the neighborhood configuration model (*NCM*) by exploring all potential topological relationships that can be distinguished by NCM and compare the result with other existing approaches.

Secondly, at the discrete level, we have designed the geometric data representations for the 3D spatial data types, which fit the compact storage in a database context and support efficient geometric algorithms for 3D operations and predicates. We have introduced a paradigm called slice representation as a general data representation method for all 3D spatial data types. The work is published in [98]. In this work, we have also presented two intersection algorithms to demonstrate the benefit of the data structures. Further, for tetrahedralized volumes, which is a popular representation in fields like computer graphics and GIS, we have designed a more efficient data structure, the connectivity encoded tetrahedral mesh (*CETM*), for storing a tetrahedralized volume object. In this article, we show that our data structure is compact in storage and is optimal in retrieving topological relations among mesh elements for both manifold volumes and non-manifold volumes. A challenging future work, is to design algorithms for the determination and evaluation of the topological predicates for spatial objects in 3D space, as well as the algorithms for evaluating cardinal directions between them.

The third level is called the implementation level, at which we integrate the data representations and the operations and predicates into database type systems and the SQL query language. We take an object oriented approach and take the **binary large object** (BLOB) as the basis for our integration. In [97], We have introduced a novel data type called the intelligent binary large object (*iBLOB*) that leverages the traditional BLOB type in databases, preserves the structure of application objects, and provides smart query and update capabilities. The iBLOB framework generates a type structure specific application programming interface (API) that allows applications to easily access the components of complex application objects. This greatly simplifies the ease with which

new type systems can be implemented inside traditional DBMS. We have implemented the iBLOB package in the database context and have built our 3D spatial data types based on it.

APPENDIX: DRAWINGS OF 50 UNIT TOPOLOGICAL RELATIONSHIP ENCODINGS AND ALL 72 TOPOLOGICAL RELATIONSHIP ENCODINGS

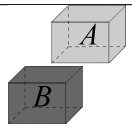
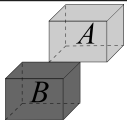
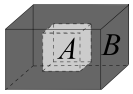
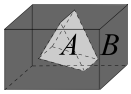
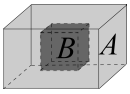
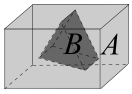
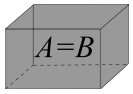
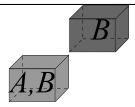
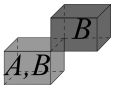
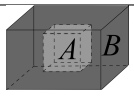
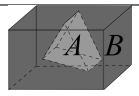
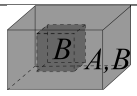
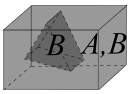
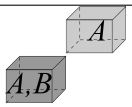
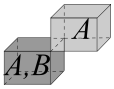
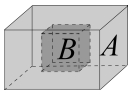
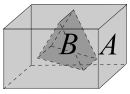
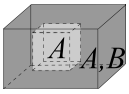
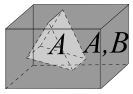
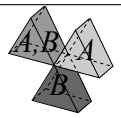
ID	1	2	3	4
				
9IM	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(011100001100000)	(011100001110000)	(011100010100000)	(011100010110000)
ID	5	6	9	10
				
9IM	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(011100011000000)	(011100011010000)	(100100100000000)	(101100100100000)
ID	11	12	13	14
				
9IM	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$
NCM	(101100100101000)	(101101000100000)	(101101000101000)	(101101100000000)
ID	15	18	19	20
				
9IM	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$
NCM	(101101100001000)	(110100101000000)	(110100101000100)	(110110001000000)
ID	17	18	19	20
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(110110001000100)	(110110100000000)	(110110100000100)	(111100101100001)

Figure A-1. 1-20 out of 50 topological relationships unit for two complex volumes.

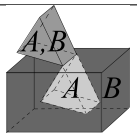
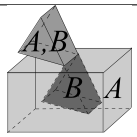
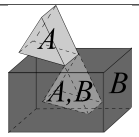
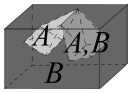
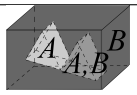
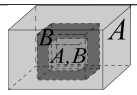
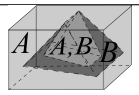
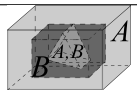
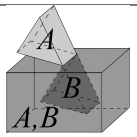
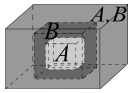
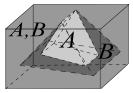
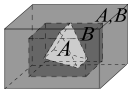
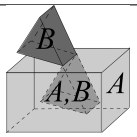
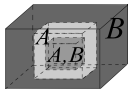
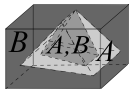
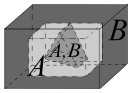
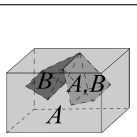
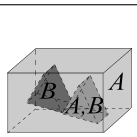
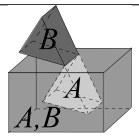
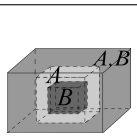
ID	43	51	75	83
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(111100110100001)	(111100111000001)	(111101001100001)	(111101010100001)
ID	84	98	99	100
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$
NCM	(111010100100010)	(111101011000000)	(111101011000001)	(111101011000010)
ID	123	146	147	148
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$
NCM	(111101101000001)	(111101110000000)	(111101110000001)	(111101110000010)
ID	235	242	243	244
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(111110001100001)	(111110010100000)	(111110010100001)	(111110010100010)
ID	251	252	283	306
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$
NCM	(111100011000001)	(111100011000010)	(111110100100001)	(111110110000000)

Figure A-2. 21-40 out of 50 topological relationships units for two complex volumes

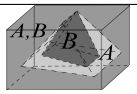
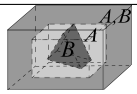
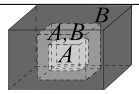
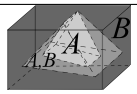
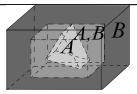
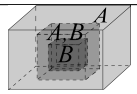
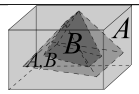
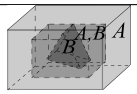
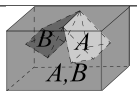
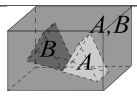
ID	307	308	394	395
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(111110110000001)	(111110110000010)	(111111000100000)	(111111000100001)
ID	396	402	403	404
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$
NCM	(111111000100010)	(111111001000000)	(111111001000001)	(111111001000010)
ID	507	508		
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$		
NCM	(111111100000001)	(111111100000010)		

Figure A-3. 41-50 out of 50 topological relationships units for two complex volumes

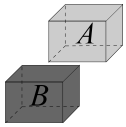
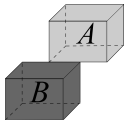
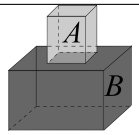
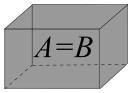
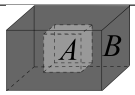
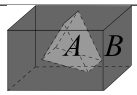
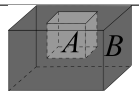
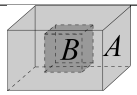
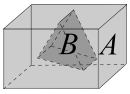
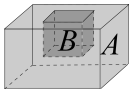
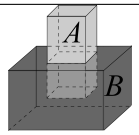
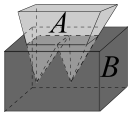
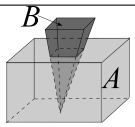
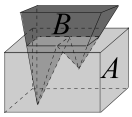
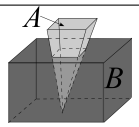
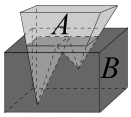
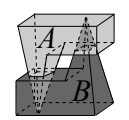
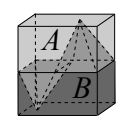
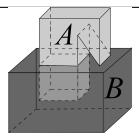
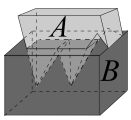
ID	1	2	8	9
				
9IM	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$
NCM	(011100001100000)	(011100001110000)	(011100011110000)	(100100100000000)
ID	12	13	17	20
				
9IM	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$
NCM	(101101000100000)	(101101000101000)	(101101100101000)	(110110001000000)
ID	21	25	411	413
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(110110001000100)	(110110101000100)	(111111001100001)	(111111001100011)
ID	415	417	419	421
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(111111001100101)	(111111001100111)	(111111001101001)	(111111001101011)
ID	423	425	427	429
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(111111001101101)	(111111001101111)	(111111001110001)	(111111001110011)

Figure A-4. 1-20 out of 72 topological relationships that can be distinguished with NCM between two simple volumes.

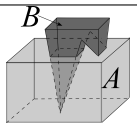
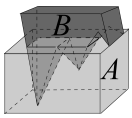
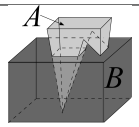
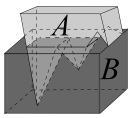
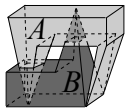
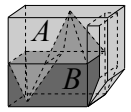
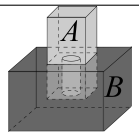
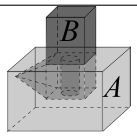
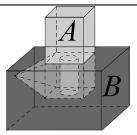
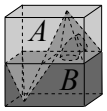
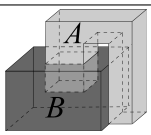
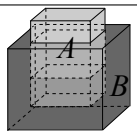
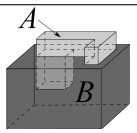
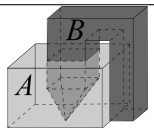
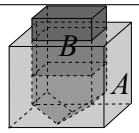
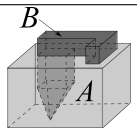
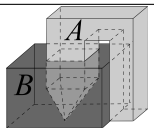
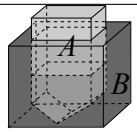
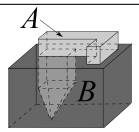
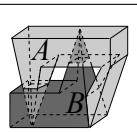
ID	431	433	435	437
				
9IM	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$
NCM	(111111001110101)	(111111001110111)	(111111001111001)	(111111001111011)
ID	439	441	477	481
				
9IM	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$
NCM	(111111001111101)	(111111001111111)	(111111011100011)	(111111011100111)
ID	485	489	491	492
				
9IM	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$
NCM	(111111011101011)	(111111011101111)	(111111011110001)	(111111011110010)
ID	493	495	496	497
				
9IM	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$
NCM	(111111011110011)	(111111011110101)	(111111011110110)	(111111011110111)
ID	499	500	501	503
				
9IM	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$	$\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix}$
NCM	(111111011111001)	(111111011111010)	(111111011111011)	(111111011111101)

Figure A-5. 21-40 out of 72 topological relationships that can be distinguished with NCM between two simple volumes.

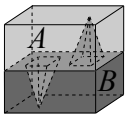
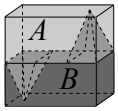
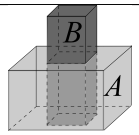
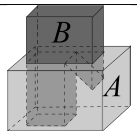
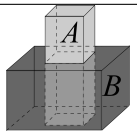
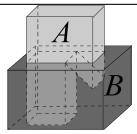
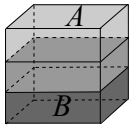
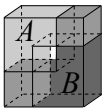
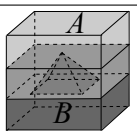
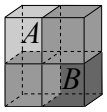
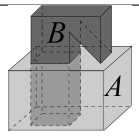
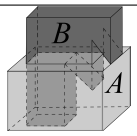
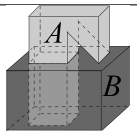
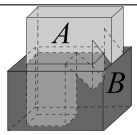
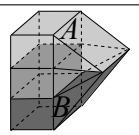
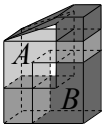
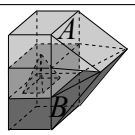
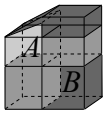
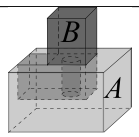
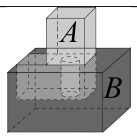
ID	504	505	559	561
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(1111110111111110)	(1111110111111111)	(1111111011100101)	(1111111011100111)
ID	563	565	566	567
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(1111111011101001)	(1111111011101011)	(1111111011101100)	(1111111011101101)
ID	568	569	575	577
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(1111111011101110)	(1111111011101111)	(1111111011110101)	(1111111011110111)
ID	579	581	582	583
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(1111111011111001)	(1111111011111011)	(1111111011111100)	(1111111011111101)
ID	584	585	673	677
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(1111111011111110)	(1111111011111111)	(1111111111100111)	(1111111111101011)

Figure A-6. 41-60 out of 72 topological relationships that can be distinguished with NCM between two simple volumes.

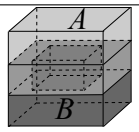
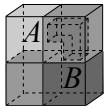
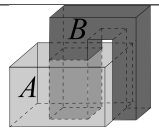
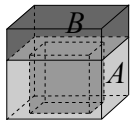
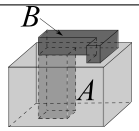
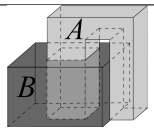
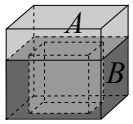
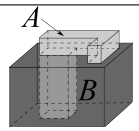
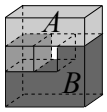
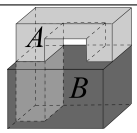
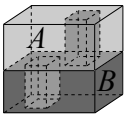
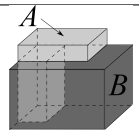
ID	680	681	687	688
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(111111111101110)	(111111111101111)	(111111111110101)	(111111111110110)
ID	689	691	692	693
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(111111111110111)	(111111111111001)	(111111111111010)	(111111111111011)
ID	694	695	696	697
				
9IM	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
NCM	(111111111111100)	(111111111111101)	(111111111111110)	(111111111111111)

Figure A-7. 61-72 out of 72 topological relationships that can be distinguished with NCM between two simple volumes.

REFERENCES

[1] Hdf-hierarchical data format. http://www.hdfgroup.org/.

[2] A. Abdul-Rahman and M. Pilouk. *Spatial Data Modelling for 3D GIS*. Springer, 2007.

[3] C Arens, J. Stoter, and P. van Oosterom. Modeling 3D Spatial Objects in a Geo-DBMS Using a 3D Primitive. *Computer & Geosciences*, 31:165–177, 2005.

[4] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority r-tree: a practically efficient and worst-case optimal r-tree. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 347–358, 2004.

[5] D. Ayala, P. Brunet, R. Juan, and I. Navazo. Object representation by means of nonminimal division quadtrees and octrees.

[6] O. Balovnev, T. Bode, M. Breunig, A.B. Cremers, W. M¨uller, G. Pogodaev, S. Shumilov, J. Siebeck, A. Siehl, and A. Thomsen. The Story of the Geotoolkit – An Object-oriented Geodatabase Kernel System. *GeoInformatica*, 8:5–47, 2004.

[7] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. Genesis: An extensible database management system. *IEEE Trans. on Software Engineering*, 14:1711–1730, 1988.

[8] B.G. Baumgart. Winged edge polyhedron representation. In *National Computer Conference*, 1975.

[9] Bruce G. Baumgart. Winged edge polyhedron representation. Technical report, Stanford University, 1972.

[10] Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 406–415, 1997.

[11] A. Biliris. The Performance of Three Database Storage Structures for Managing Large Objects. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 276–285, 1992.

[12] Urs Bischoff and Jarek Rossignac. Tetstreamer: Compressed back-to-front transmission of delaunay tetrahedra meshes. In *Proceedings of the Data Compression Conference*, pages 93–102, 2005.

[13] I. Boada. An octree-based multiresolution hybrid framework. *Future Gener. Comput. Syst.*, 20(8):1275–1284, 2004.

[14] A. Borrmann, C. van Treeck, and E. Rank. Towards a 3D Spatial Query Language for Building Information Models. In *Proceedings of the Joint International Conference for Computing and Decision Making in Civil and Building Engineering*, 2006.

[15] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0. *W3C recommendation*, 6, 2000.

[16] S. F. Buchele and R. H. Crawford. Three-dimensional halfspace constructive solid geometry tree construction from implicit boundary representations. In *Symposium on Solid Modeling and Applications*, pages 135–144, 2003.

[17] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed edgesa scalable representation for triangle meshes. *J. Graph. Tools*, 3:1–11, 1998.

[18] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A Data Model and Query Language for Exodus. *ACM SIGMOD Record*, 17:413 – 423, 1988.

[19] Dan Chen, Yi-Jen Chiang, Nasir Memon, and Xiaolin Wu. Geometry compression of tetrahedral meshes using optimized prediction. In *IN PROC. EUROPEAN CONFERENCE ON SIGNAL PROCESSING*, 2005.

[20] J. Chen, D. Liu, H. Jia, and C. Zhang. Cardinal Direction Relations in 3D Space. In *Int. Conf. on Knowledge Science, Engineering and Management*, pages 623–629, 2007.

[21] T. Chen, H. Liu, and M. Schneider. Evaluation of Cardinal Direction Developments between Moving Points. In *ACM Symp. on Geographic Information Systems (ACM GIS)*, pages 430–433, 2010.

[22] T. Chen, H. Liu, and M. Schneider. Computing the Cardinal Direction Development between Moving Points in Spatio-temporal Databases. In *The International Symp. on Spatial and Temporal Databases (SSTD) (Accepted)*, 2011.

[23] T. Chen and M. Schneider. Modeling Cardinal Directions in the 3D Space with the Objects Interaction Cube Matrix. In *ACM Symp. on Applied Computing (ACM SAC)*, pages 906–910, 2010.

[24] Paolo Cignoni, Leila De Floriani, Paola Magillo, Enrico Puppo, and Roberto Scopigno. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics*, 10:29–45, 2004.

[25] E. Clementini, P. D. Felice, and P. Oosterom. A Small Set of Formal Topological Relationships Suitable for End-user Interaction. In *3thInt. Symp. on Advances in Spatial Databases*, pages 277–295, 1993.

[26] B. de Cambray and T s. Yeh. A Multidimensional (2D, 2.5D, and 3D) Geographical Data Model. In *6th International Conference on Management of Data*, 1994.

[27] Leila De Floriani, David Greenfieldboyce, and Annie Hui. A data structure for non-manifold simplicial d-complexes. In *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 83–92, 2004.

[28] Leila De Floriani and Annie Hui. A scalable data structure for three-dimensional non-manifold objects. In *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 72–82, 2003.

[29] Leila De Floriani and Annie Hui. Data structures for simplicial complexes: an analysis and a comparison. In *Proceedings of the 3rd Eurographics symposium on Geometry processing*, 2005.

[30] Laszlo M. Dobkin D. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 5:3–32, 1989.

[31] M. J. Egenhofer. Topological Relations in 3D. Technical report, 1995.

[32] M. J. Egenhofer and J. Herring. A Mathematical Framework for the Definition of Topological Relationships. In *Int. Symp. on Spatial Data Handling*, pages 803–813, 1990.

[33] M. J. Egenhofer and J. Herring. Categorizing Binary Topological Relations between Regions, Lines and Points in Geographic Databases. Technical Report Technical Report, 1990.

[34] M. Erwig, R. Hartmut Guting, M. Schneider, and M. Vazirgiannis. Spatio-temporal Data Types: an Approach To Modeling and Querying Moving Objects in Databases. *GeoInformatica*, 3(3):269–296, 1999.

[35] ESRI. Esri shapefile technical description, 1998. http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf.

[36] L. De Floriani, R. Fellegara, and P. Magillo. Design, Analysis and Comparison of Spatial Indexes for Tetrahedral Meshes. Technical report, 2010.

[37] L. De Floriani and A. Hui. A Scalable Data Structure for Three-dimensional Non-manifold Objects. In *Eurographics Symposium on Geometry Processing*, pages 72–82, 2003.

[38] J. D. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer graphics, principles and practice*. Addison-Wesley, 1990.

[39] L. Forlizzi, R. Guting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 319–330, 2000.

[40] A. Frank. Qualitative Spatial Reasoning: Cardinal Directions As an Example. *International Journal of Geographical Information Science*, 10(3):269–290, 1996.

[41] S. Gaal. *Point Set Topology*. Academic Press, 1964.

[42] S. Gaal. *Point Set Topology*. Academic Press, 1964.

[43] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30:170–231, 1998.

[44] R. Goyal and M. Egenhofer. Cardinal Directions between Extended Spatial Objects. Unpublished manuscript, 2000.

[45] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. on Graphics*, 4:74–123, 1985.

[46] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4:74–123, 1985.

[47] Stefan Gumhold, Stefan Guthe, and Wolfgang Straber. Tetrahedral mesh compression with the cut-border machine. In *Proceedings of the conference on Visualization*, pages 51–58, 1999.

[48] W. Guo, P. Zhan, and J. Chen. Topological Data Modeling for 3D GIS. *Int. Archives of Photogrammetry and Remote Sensing*, 32(4):657–661, 1998.

[49] Topraj Gurung and Jarek Rossignac. Sot: compact representation for tetrahedral meshes. In *SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pages 79–88, 2009.

[50] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57, 1984.

[51] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey, and E. J. Shekita. Starburst mid-flight: As the dust clears. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 2:143–160, 1990.

[52] J.n Huerta, M. Chover, J. Ribelles, and R. Quiros. Multiresolution modeling using binary space partitioning trees. *Computer Networks*, 30(20-21):1941–1950, 1998.

[53] B. Hwang, I. Jung, and S. Moon. Efficient storage management for large dynamic objects. In *EUROMICRO 94. System Architecture and Integration 20th EUROMICRO Conference.*, pages 37–44, Sep 1994.

[54] A. Safonova J. Rossignac and A. Szymczak. Edgebreaker on a corner table: A simple technique for representing and compressing triangulated surfaces. In *Hierarchical and Geometrical Methods in Scientific Visualization*, pages 41–50, 2003.

[55] J. Legakis K. I. Joy and R. MacCracken. Data structures for multiresolution representation of unstructured meshes. In *Hierarchical Approximation and Geometric Methods for Scientific Visualization*, 2002.

[56] Marcelo Kallmann and Daniel Thalmann. Star-vertices: A compact representation for planar meshes with adjacency information. *J. Graph. Tools*, 6:7–18, 2002.

[57] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *Proceedings of the second international conference on Information and knowledge management*, CIKM '93, pages 490–499, 1993.

[58] C. Tet Khuan, A. Abdul-Rahman, and S. Zlatanova. 3D Solids and Their Management in DBMS. In *Advances in 3D Geoinformation Systems*, pages 279–311, 2008.

[59] Bryan Matthew Klingner, Department of Electrical Engineering Jonathan Richard Shewchuk, and Berkeley Computer Sciences University of California. Stellar: A tetrahedral mesh improvement program, 2009. http://www.cs.berkeley.edu/~jrs/stellar/.

[60] Marcos Lage, Thomas Lewiner, Helio Lopes, and Luiz Velho. Chf: A scalable topological data structure for tetrahedral meshes. In *In Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing*, pages 349–356, 2005.

[61] J. Lee. Comparison of existing methods for building triangular irregular network models of terrain from grid digital elevation models. *International Journal of Geographical Information Systems*, 5(3):267–285, 1991.

[62] Pascal Lienhardt. Topological models for boundary representation: a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23:59–82, 1991.

[63] X. Liu, W. Liu, and C. Li. Qualitative Representation and Reasoning of Combined Direction and Distance Relationships in Three-dimensional GIS. In *Proceedings of the International Conference on Computer Science and Software Engineering*, pages 119–123, 2008.

[64] Hélio Lopes and Geovan Tavares. Structural operators for modeling 3-manifolds. In *Proceedings of the 4th ACM Symposium on Solid Modeling and Applications*, pages 10–18, 1997.

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

[66] M. Schneider, T. Chen, G. Viswanathan and W. Yuan. Cardinal Directions between Complex Regions. *ACM Transactions on Database Systems (TODS) (submitted)*, In press, 2011.

[67] Martti Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Inc., New York, NY, USA, 1987. ISBN 0-88175-108-1.

[68] Research Group: Numerical Mathematics, Scientific Computing Weierstrass Institute for Applied Analysis, and Stochastics (WIAS). Tetgen, 2011. http://tetgen.org.

[69] R.E. McGrath. Xml and scientific file formats. In *The Geological Society of America*, 2003.

[70] A. E. Middleditch, C. M. P. Reade, and A. J. Gomes. Point-sets and cell structures relevant to computer aided design. *Int. Journal of Shape Modeling*, 6(2):175–205, 2000.

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

[72] 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.

[73] Preparata F. P. Muller D. E. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.

[74] B. Naylor. Binary space partitioning trees as an alternative representation of polytopes. *Computer-Aided Design*, 22(4):250–252, 1990.

[75] Gregory M. Nielson. Tools for triangulations and tetrahedrizations. In *Scientific Visualization, Overviews, Methodologies, and Techniques*, pages 429–525, 1997.

[76] University of Minnesota. Geomview, 1998. http://www.geomview.org/.

[77] A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci. Dimension-independent modeling with simplicial complexes. *ACM Transactions on Graphics (TOG)*, 12: 56–102, 1993.

[78] D. Papadias, Y. Theodoridis, and T. Sellis. The Retrieval of Direction Relations Using R-trees. In *Int. Conf. on Database and Expert Systems Applications (DEXA)*, pages 173–182, 1994.

[79] Stratos Papadomanolakis, Anastassia Ailamaki, Julio C. Lopez, Tiankai Tu, David R. O'Hallaron, and Gerd Heber. Efficient query processing on unstructured tetrahedral meshes. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 551–562, 2006.

[80] F. Penninga and P. Van Oosterom. A Simplicial Complex-based DBMS Approach to 3D Topographic Data Modelling. *Int. J. Geogr. Inf. Sci.*, 22(7):751–779, 2008.

[81] F. Penninga and P. van Oosterom. First Implementation Results and Open Issues on the Poincare-ten Data Structure. In *Advances in 3D Geoinformation Systems*, pages 177–197, 2008.

[82] S. Pigot. Topological Models for 3D Spatial Information Systems. In *International Conference on Computer Assisted Cartography(Auto-Carto)*, pages 368–392, 1991.

[83] J. F. Raper and D. J. Maguire. Design models and functionality in gis. *Comput. Geosci.*, 18(4):387–394, 1992.

[84] Princeton Shape Retrieval and Analysis Group. The princeton shape benchmark, 2005. http://shape.cs.princeton.edu/benchmark/.

[85] R.K. Rew, B. UCAR, and EJ Hartnett. Merging netcdf and hdf5. In *20th Int. Conf. on Interactive Information and Processing Systems*, 2004.

[86] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 0123694469.

[87] H.-J. Schek, H.-B. Paul, M. H. Scholl, and G. Weikum. The DASDBS Project: Objectives, Experiences, and Future Prospects. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 2(1):25–43, 1990.

[88] M. Schneider and T. Behr. Topological Relationships between Complex Spatial Objects. *ACM Trans. on Database Systems (TODS)*, 31(1):39–81, 2006.

[89] 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.

[90] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 507–518, 1987.

[91] S. Shekhar, X. Liu, and S. Chawla. An Object Model of Direction and Its Implications. *GeoInformatica*, 3(4):357–379, 1999.

[92] Jonathan Richard Shewchuk. Tetrahedral mesh generation by delaunay refinement. In *Proceedings of the 14th annual symposium on Computational geometry*, pages 86–95, 1998.

[93] Hang Si and Klaus Gartner. Meshing piecewise linear complexes by constrained delaunay tetrahedralizations. In *In Proceedings of the 14th International Meshing Roundtable*, pages 147–163. Springer, 2005.

[94] N. Stewart, G. Leach, and S. John. An improved z-buffer csg rendering algorithm. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 25–30, 1998.

[95] M. Stonebraker. Inclusion of new types in relational data base systems. In *Int. Conf. on Data Engineering Conference (ICDE)*, pages 262–269, 1986.

[96] Andrzej Szymczak and Jarek Rossignac. Grow & fold: compression of tetrahedral meshes. In *Proceedings of the 5th ACM Symposium on Solid Modeling and Applications*, pages 54–64, 1999.

[97] T. Chen, A. Khan, M. Schneider and G. Viswanathan. *iBLOB*: Complex Object Management in Databases Through Intelligent Binary Large Objects. In *3rd Proceedings of the international conference on Objects and databases*, pages 85–99, 2010.

[98] T. Chen and M. Schneider. Data Structures and Intersection Algorithms for 3D Spatial Data Types. In *17th ACM Symp. on Geographic Information Systems (ACM GIS)*, pages 148–157, 2009.

[99] T. Chen and M. Schneider. The Neighborhood Classification Model: a New Framework To Distinguish Topological Relationships between Complex Volumes. In *Proceedings of the 30th international conference on Advances in conceptual modeling: recent developments and new directions*, ER'11, pages 251–260, 2011.

[100] T. Chen, M. Schneider, G. Viswanathan and W. Yuan. The Objects Interaction Matrix for Modeling Cardinal Directions in Spatial Databases. In *International Conference on Database Systems for Advanced Applications*, pages 218–232, 2010.

[101] R. B. Tilove. Set Membership Classification: a Unified Approach To Geometric Intersection Problems. *IEEE Transactions on Computers*, 29:874–883, 1980.

[102] R. O. C. Tse and C. M. Gold. Tin meets cad - extending the tin concept in gis. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part III*, pages 135–143, 2002.

[103] P. van Oosterom, W. Vertegaal, M. van Hekken, and T. Vijlbrief. Integrated 3D Modeling Within a GIS. In *Int. Workshop on Advanced Geographic Data Modelling*, pages 80–95, 1994.

[104] R. Weibel and M. Heller. Digital terrain modelling. *Geographical Information Systems*, 1:269–297, 1991.

[105] K. Zeitouni, B. De Cambray, and T. Delpy. Topological Modelling for 3D GIS. In *4thInt. Conf. on Computers in Urban Planning and Urban Management*, 1995.

[106] S. Zlatanova. On 3D Topological Relationships. In *Int. Conf. on Database and Expert Systems Applications (DEXA)*, page 913, 2000.

[107] S. Zlatanova, P. Mathonet, and F. Boniver. The Dimensional Model: A Framework To Distinguish Spatial Relationships. In *Int. Symp. on Advances in Spatial Databases*, pages 285–298, 2002.

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

BIOGRAPHICAL SKETCH

Tao Chen has received his B.S. from Nanjing University in China in 2006. He started his study in the University of Florida in 2006. His research focuses on spatial database modeling, especially on the design of data structures and algorithms for 3D data management in databases. He received his Ph.D. from the University of Florida in the summer of 2012.