Concepts for the Representation, Storage, and Retrieval of Spatio-Temporal Objects in 3D/4D Geo-Information-Systems

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Jörg Siebeck

aus

Düsseldorf

Bonn 2003

Concepts for the Representation, Storage, and Retrieval of Spatio-Temporal Objects in 3D/4D Geo-Information-Systems

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) within the Graduiertenkolleg 437 "Das Relief – eine strukturierte und veränderliche Grenzfläche". The support is gratefully acknowledged.

Contents

Chapter 1

Introduction

The requirements for the modelling of time are manifold in geoscientific applications. They are covering time conceptions like the modelling of discrete time stamps or the management of dynamic geo-processes. The fact that entities of the dynamic environment may continously change both location and shape makes their modelling a complex task.

The work presented has its root in interdisciplinary projects hosted at Bonn University with participants from the fields of the geo-sciences, computer science, and mathematics. In particular, one class of applications aims at supporting the database management of geo-scientific entities. The requirements, though, are demanding. For example, geologically defined geometries are inherently three dimensional, and over time their change may be modelled in a continuous way. A GIS-in-a-box-package that copes with these two requirements seems to be not available at present.

Indeed, extending database technology by facilities for storing and querying spatio-temporal data has become an active field of research over the last years; however, the management of such time-dependent geometries remains a challenge for the database community (Sellis, 1999). On the one hand, the enormous growth of raw data caused by the introduction of the fourth dimension requires a maximally compact storage of changing objects. On the other hand, data analysis demands that the database system possess interactive query processing facilities which range from selections for online animation to complex set-based operations such as spatio-temporal joins (Güting et al., 2000).

Geo-data handling is embedded in an information processing environment. A *geo-information system* (GIS) includes the tasks of an information system and extends them. A GIS can thus be characterised as “a computer-based information system that enables capture, modelling, manipulation, retrieval, analysis and presentation of geo-referenced data” (Worboys, 1995, p. 1, altered). Usually, the main tasks of *storage* and *manipulation*—retrieval and update—of data are put under control of a specialised software system, the so-called *database management system* (DBMS), or database system for short. A DBMS that offers special facilities for the handling of spatial data is called a *spatial DBMS* (Rigaux et al., 2002; Güting, 1994; Breunig, 2001).

Investigating the *dynamic* aspects of environment has since long been recognised as an important research need. More recently, Peuquet and Qian (1996) observe that "the ability to examine the *dynamics* of geographic phenomena is urgently needed as an essential tool for examining and increasing our understanding of man-environment interactions at local, regional, and global scales". Parent et al. (1999, p. 3) emphasise this point, who argue that applications also need to express that some information varies in space and/or time and that any attribute can be time-varying. Zhang and Hunter (2000) observe that "most of the research has focused on the treatment of discrete changes in spatial objects, for example in cadastral parcels. However less attention has been paid to the treatment of continuous change which occurs primarily in dynamic objects found in the natural environment, for example in seasonal coastal changes". Some of the requirements when dealing with temporal aspects of *terrain modelling* are given by Van Kreveld (1997, p. 71):"Suppose that a sequence of terrains is given, each representing the terrain at a fixed moment in time. [...] Since the data usually is available only at a discrete set of moments, interpolation between two terrains at consecutive moments becomes necessary. [...] One more use of TINs[1] in geographic processing is the simulation of physical processes that influence certain terrain features."

If a GIS also handles such *temporal variation* in geo-referenced data, the GIS is termed a *temporal geo-information system* (Abraham and Roddick, 1999; Claramunt et al., 1997; Peuquet and Qian, 1996; Zhang and Hunter, 2000), a *time-integrative GIS* (Ott and Swiaczny, 2001) or a *4D-GIS* (in case of three spatial and one temporal dimension, see also Breunig, 2001). The particular role of temporal GIS (TGIS) lies in the tracing of the lineage of spatial objects and their attributes (Abraham and Roddick, 1999, chap. 2). Besides the spatial 'where' and the thematic 'what' queries a geo-database must support a 'when'-operation (Voigtmann et al., 1996, p. 7).

While spatial DBMSs are an adequate tool for the data storage component within a GIS, it is questionable whether they are also adequate for temporal GIS. As the semantics of the spatial and temporal dimensions, which are intrinsic properties of the data, are unknown to the underlying DBMS, spatio-temporal aplications do not currently enjoy the built-in, integrated support that current DBMS's supply to less challenging applications (Böhlen et al., 1998, sect. 1). The same point is observed by Wolfson et al. (1998, sect. 1): "Existing DBMS's are not well equipped to handle continuously changing data, such as the location of moving objects. The reason for this is that in databases, data is assumed to be constant unless it is explicitly modified." The particular problem of continuous change of spatial data has also been emphasised by D. Peuquet, when she has put it into the context of the field of temporal database systems. She wrote (Peuquet, 2001, p. 11, abstract): "Even with much activity over the past decade, including organized efforts on both sides of the Atlantic, the representation of both space and time in digital databases is still problematic and functional space-time systems have not gone beyond the limited prototype stage." And furthermore, (Peuquet, 2001, pp. 16, bottom): "Current temporal DBMS can handle discrete change, but continuous change can only be implied from one recorded state to the next. [...] However, most geographic spatio-temporal applications involve continuous change of objects in space." Despite the long tradition of both temporal and spatial databases in computer science, the integration of both fields is only in its infancy (Egenhofer et al., 1999, pp. 787). However, computerised geo-information systems (GIS), though now widespread within the geo-sciences, are not well equipped with the necessary capabilities for an adequate support. Egenhofer

[1]TIN = triangular irregular network

Basic STP	Geometric types	Shape	Size	Orientation	Location
Stability	any	const.	const.	const.	const.
Deformation	any but point	**changed**	const.	const.	const.
Expansion	any	const.	**growing**	const.	const.
Contraction	any but point	const.	**shrinking**	**changed**	const.
Rotation	any but point	const.	const.	**changed**	const.
Translation	any	const.	const.	const.	**changed**

Table 1.1: Taxonomy of basic spatio-temporal processes (STPs) according to Claramunt et al. (1997).

et al. (1999, pp. 787) note that "most of today's computational methods in GIScience treat geographical phenomena as static. A variety of conceptual models for time in GIS have been studied [...], but to date little impact has been made on commercially available tools."

Usually, the term '*spatio-temporal data*' is adopted in database and GIS literature when dealing with both temporal and spatial aspects. Unfortunately, a clear definition of what kind of data a spatio-temporal database system is expected to support is lacking. Table 1.1 shows a taxonomy of basic spatio-temporal processes according to Claramunt et al. (1997), which gives an impression in which complex ways processes act upon geo-objects. It is obvious that a spatio-temporal DBMS must be able to represent the results of these processes, if it aims at a larger class of applications. Seen from a different perspective, a classification scheme could help identifying what aspects are supported in a spatio-temporal database system. One frequently cited example of such a classification scheme appears in the work of Theodoridis et al. (1998). Although formulated for the purpose of indexing, it is used in a wider context here. This classification is purely pragmatic, but useful for the discussion of this work's requirements and of what particular kind of spatio-temporal data it aims at. Therefore, in the following selected specifications of Theodoridis et al. (1998) are discussed here, while they are also extended and slightly revised where appropriate. Figure 1.1 gives a short reference of these specifications.

The first specification relates to the *time dependency* of spatial objects. (This specification is not among those of Theodoridis et al. (1998).) Given that a spatio-temporal object can be regarded as a function f from the temporal domain into the spatial domain, three important classes can be distinguished. Firstly, there is the special case of a *step-wise constant* function. Hence, a spatio-temporal object is given by associating a spatial object o with a time stamp T with the meaning that for each $t \in T : \ f(t) = o$. Secondly, there is the special case of a function that changes an object's *position* only. Hence, a spatio-temporal object is given by associating a spatial object o with a motion function m with the meaning that for each $t \in T : \ f(t) = \bigcup_{p \in o} m(p)$. Thirdly—and most general—, there is the special case of functions that change both *position and shape* of a spatial object. This specification is independent of the other specifications given below and is well suited to subsume most models introduced in related work. Obviously, complexity increases with the three classes mentioned. The first class of step-wise constant functions is at the heart of (traditional) temporal database models. If a system combines a spatial and a temporal model without any further ado, it falls into this category, for instance Böhlen et al. (1998). The second class of position-changing functions is interesting for certain applications, for example, applications that track the position of mobile objects (Wolfson, 2002; ESRI, 1998). It is worth noting that there is an inclusion relationship between the three classes. This thesis falls into the third category, since the other categories are not expected to be well suited for the complex dynamic objects of environment.

The second specification relates to the *spatial data types* that are supported. Originally, this specification distinguished between systems that deal with *point* data only and systems that also include *region* data. Here, this specification is reformulated as having the categories *dimension=0*, *dimension=0..2*, and *dimension=0..3*. The previous specification has been changed here, since the majority of related work deals explicitly with objects of spatial dimensions 0..2 embedded in $\mathbb{R}^2$, whereas also three-dimensional objects in $\mathbb{R}^3$ are of interest. Strictly, this specification is not orthogonal to the previous one, since changing shape of a 0-dimensional object is not meaningful. This thesis falls into the third category. It is worth noting that the distinction between spatial and spatio-temporal data will be removed here (at least on an application programming interface level, see also chapter 4).

While the previous specification dealt with the spatiality of objects, the next one relates to the temporality of objects and asks for which *time dimensions* are supported. This specification distinguishes between the traditional temporal domains *valid time*, *transaction time* and the combined *bitemporal* domain. The first and the second domain are used to model the time when facts are current in the real world and in the database system, respectively. The third domain is the Cartesian product of valid and transaction time. This thesis does not account for transaction time of objects and falls therefore into the first category of valid time.

The next specification relates to the *data set mobility*. It aims at the number of spatio-temporal objects that have to be supported and distinguishes between: (1) a *growing/shrinking* data set, where the number of spatio-temporal objects changes over time; (2) an *evolving-only* data set where the number of objects remains fixed; (3) a *fully dynamic* environment that includes both kinds of data sets. It must be noted that the specifications of Theodoridis et al. (1998) are targeted at indexing, also for restricted classes of applications. However, since an evolving-only data set must be ruled out for a general class of applications, such restrictions cannot be justified for the scope of this thesis.

The next specification asks whether past database states can be subject to change, that means the specification is *with respect to time-stamp update*. It distinguishes between the cases when database objects are loaded once and not changed thereafter (*static case*), when only current states of objects are allowed to change (*chronological case*), and the case when also past states of objects are allowed to change (*dynamic case*). The static case could be of interest when dealing solely with results of simulation or similar entities for which a subsequent change is of no sense. The chronological case is interesting for, e.g., monitoring applications. For the scope of this thesis, however, the third case ("dynamic") is central, since it occurs when spatio-temporal objects are subject to a geometric modelling task (see also section 3.1.2).

The two following specifications given by Theodoridis et al. (1998) aim solely at the problem of indexing and are not discussed here: (1) Handling "obsolete" entries through support of bulk-loading, packing, or purging operations; (2) a specific object approximation to be used for indexing. Finally, a specification asks for *specific, application-oriented query processing operations*. This yes/no-specification leads to the classes of algorithms and query

Specification	Explanation (domain)
0. Time dependency of spatial objects	step-wise-constant | function-of-location | function-of-location-and-shape
1. Spatial data types supported	dimension = 0 | dimension = 0..3
2. Database classification I: with respect to time dimension(s) supported	valid-time | transaction-time | bitemporal
3. Database classification II: with respect to data set mobility	growing | evolving | full-dynamic
4. Database classification III: with respect to time-stamp update	static | chronological | dynamic
5. Specific object approximation	No | yes
6. Handling "obsolete" entries (e.g. support of bulk-loading/packing/purging operations)	No | yes
7. Specific (application-oriented) query processing operations	No | yes

Figure 1.1: Classification scheme for spatio-temporal indexing as reported by Theodoridis et al. (1998), changed: Specification 0 introduced, specification 1 altered.

operators that are to be supported seen from an application's level. For the scope of this thesis, these operators are specified in chapters 4 and 6.

The benefits pertaining DBMS-usage within an information system should carry over to TGIS. To this end, also spatio-temporal data handling must be brought under control of the DBMS and, hence, its main responsibilities must be applied to this novel kind of data: storage and manipulation.

A prerequisite of storage is representation. However, a mismatch concerning spatio-temporal data and its integration into databases has been observed by many researchers (e.g. Peuquet, 2001; Böhlen et al., 1998). In particular, Parent et al. (1999) state that "there is a mismatch between the logical, implementation-oriented view of data supported by the tools, and the application-oriented, conceptual view that users follow in their everyday work". This fact has initiated several research efforts. For instance, in MADS (modeling of application data with spatio-temporal features, Spaccapietra et al. (1998)) the database designer is equipped with a set of modelling constructs, among which are (1) well-known features such as objects, attributes, methods, integrity constraints, n-ary relationships, is-a links, and aggregation links; (2) spatial abstract data types; (3) the option of adding time-variability to the above mentioned entities, in particular to a spatial object: "Specifying a spatial attribute as time-varying allows to describe objects that move and change their shape." (Parent et al., 1999). However, the gap between the conceptual level and the implementation level still remains.

Technically, the requirement of representing spatio-temporal data on the *implementation's level* can be met by three principal approaches. They match those found in representing purely spatial data within databases as reported, e.g., by Rigaux et al. (2002).

1. Ontology, structure, and representation of space and time.
2. Models and languages for spatio-temporal database systems.
3. Graphical user interfaces for spatio-temporal information.
4. Query processing in spatio-temporal databases.
5. Storage structures and indexing techniques for spatio-temporal databases.
6. The architecture of a spatio-temporal database systems.

Figure 1.2: According to Sellis (1999), research issues in spatio-temporal database systems still cover a wide range, as this list indicates.

Firstly, spatio-temporal entities on the conceptual level can be mapped onto a DBMS data model, e.g., relational. A prominent example for two-dimensional purely spatial data is the OGC *Simple Feature Specification for SQL*. Similarly, Spaccapietra et al. (1998) developed methods to translate the modelling constructs of MADS (see above) to the data model of TSQL2 (Snodgrass et al., 1994). The advantage of this overall approach is flexibility, since spatio-temporal types can be modelled quite freely. However, the approach suffers from several drawbacks, for instance missing data independence.

Secondly, incorporating spatio-temporal data handling into a DBMS can be performed by using a specialised data store. Spatio-temporal entities on the conceptual level can be mapped onto the data structures of the specialised data store. A prominent example for the pure-spatial domain is the ARC-Storm data manager for spatial data (ESRI, 1996). Positively, the drawback of missing data independence is eliminated in this approach. However, spatio-temporal data and thematic data remain separated and cannot be processed in an integrated way.

The third option of putting spatio-temporal data handling under control of a DBMS lies in extending an existing data model with the appropriate data types that can then be used as a mapping target from the conceptual to the logical level. A prominent example—again for two-dimensional purely spatial data—is the OGC Simple Feature Specification for SQL using spatial data types. This approach rules out the disadvantages of the previous approaches and offers a tight integration into all components of a DBMS.

To summarise, not only has the third approach advantages concerning *representation* of spatio-temporal data, but also concerning its *manipulation*. Spatio-temporal operations can be embedded into the DBMS and, hence, used in queries or integrity constraints. Methods for indexing can be integrated that support efficient query processing. This tight integration into a DBMS is not possible with the two other approaches, and, as a consequence, this thesis follows the last approach in that it offers tight integration into a DBMS.

Several researchers have addressed the question of what research work is needed to facilitate the aforementioned tight integration with a DBMS. Figure 1.2 lists the research issues that have been identified by Sellis (1999). Abraham and Roddick (1999, chap. 6) stress the aspect of the high storage space utilisation when time is introduced as a further dimension. The authors argue that the unmanageable storage cost can be dealt with, if new and innovative data structures with minimal storage requirements are developed. Furthermore, through the high storage cost data retrieval becomes problematic, such that advanced querying, visualisation, and analytical reasoning abilities must be introduced into the DBMS (Abraham and Roddick, 1999, chap. 6).

The contribution of the work presented here is threefold. First, it is investigated how spatio-temporal data can be described through a *representational model* for continuously changing spatial data. A model is introduced that evolves from previous models developed in our former research projects. The model extends the well-known simplicial complex model for spatial data by time in that it allows for continuous as well as discontinuous change. As a negative result, the non-closure under certain operations is shown. The model is mapped onto a small system of classes in an object-oriented database framework. Second, internal data structures are introduced that represent instances of the (user-level) spatio-temporal classes. A new technique provides a compromise between compact storage, flexibility, and efficient retrieval of spatio-temporal objects. These structures correspond to temporal graphs and support updates as well as the maintainance of connected components over time. Third, it is shown how to realise further operations on the new type of objects. Among these operations are range queries, intersection tests, and the Euclidean distance function.

The remainder of the thesis is organised as follows. The following chapter introduces related work in the field of spatio-temporal database systems. Then, in the next chapter the new representational model for spatio-temporal objects is introduced. Chapter 4 develops objectives for the integration of the new types of objects into an object-oriented database framework and introduces a small system of classes. Chapter 5 presents implementation concepts for these classes. Chapter 6 shows how fundamental query operations can be computed for the new classes of spatio-temporal objects. Finally, chapter 7 concludes and gives an outlook for future work on the thesis topics.

Chapter 2

Related Work

A concensus representational model for spatio-temporal data has not yet emerged. This chapter gives an overview of the many models that have been presented. Operations and query capabilities of these models are discussed. Less work is available for storage structures that implement these model. An exception is the large amount of work on index structures for spatio-temporal data, an overview of which is given in chapter 5.5.

Some of the research work on spatio-temporal database systems originated in the field of temporal databases (Tansel et al., 1993; Jensen et al., 1998). The rationale behind these approaches is often such that the facility of a temporal data model to capture time-varying data implies the facility to capture time-varying *spatial* data as a special case—assuming spatial abstract data types are available. As a representative example, Bertino et al. (1998) model a time-dependent object o as a *partial function* from the time-domain into the set of legal values for o. This approach is at the heart of an extension to the ODMG object model, called T_ODMG (Bertino et al., 1998). In T_ODMG, the time-domain is isomorphic to the set of natural numbers, hence it is discrete and linear. This model directly supports the temporal development of objects. One can declare a time-dependent variable of type `t` with the aforementioned semantics by the syntax `temporal(t)`. It is worth noting that `temporal()` is not an orthogonal type constructor in T_ODMG, since it yields a type that cannot be an argument to `temporal()`. Embedding spatio-temporal data handling in this framework seems possible by giving `temporal(`*`spatial`*`)` a proper implementation, while reusing the interface of T_ODMG. However, no query facilities have been defined in (Bertino et al., 1998).

Such temporal developments of object properties are also focus of the Tripod project, where they are called *histories*. The *Tripod* project aims at developing a spatio-temporal object database system that extends the ODMG standard for object models (Griffiths et al., 2001c). This system is targeted at *discrete* changes to data that— as a special case —can also be spatial. The spatial domain of this system is built on the ROSE algebra of spatial data types (Güting and Schneider, 1995). Hence, Tripod is capable of representing points, lines and polygons, each on a discrete grid as the underlying geometric domain, the so-called *realm*. The operators of the ROSE algebra are supported. In ODMG terms the spatial types are *literals*, i.e., values without an object identifier.

Tripod's temporal domain is also based on realms, restricted to one dimension (Griffiths et al., 2002). As a consequence, this domain is a finite subset of the set of natural numbers. It is linear and totally ordered. The distance between two adjacent points in this domain is a *chronon.* The model comprises four temporal types: (1) a type representing an instant, called *instant*: a point in the temporal domain; (2) a type for a temporal interval, called *timeInterval* that is given by a starting and ending point from the temporal domain; (3) a type for a set of intervals, called *timeIntervals*; and (4) a type for a set of instants, called *instants.* A *timestamp* is either a value of type *instants* or of type *timeInterval.* Temporal intervals in this model are always *closed*: the starting and the ending instant are both members of the interval. The *duration* of a *timeInterval* is at least two chronon, since $start(i) < end(i)$ is an integrity constraint on each interval i (Griffiths et al., 2002).

Two special instants *beginning* and *forever* are offered as literals and represent the earliest and latest representable values. The literal *until_changed* specifies a fact valid at present, but that can change at some future instant. During query processing, every occurrence of this literal is replaced by the value of "now."

Predicates are available for testing binary relationships among the aforementioned temporal types. The authors state that predicates from other approaches, e.g., Allen (1983), are either directly supported or can be built upon the Tripod-predicates (Griffiths et al., 2002). Among these predicates are:

- Predicates defined on the primitive types *instant* and *timeInterval*: equals, before, overlap, meet, disjoint, on, in, contains. Since the exact definitions are straightforward, they are omitted here (see Griffiths et al. (2002) for details).
- Extended versions of these predicates for the collection-valued types *instants* and *timeIntervals*: Each predicate θ is available in the form $\theta\_\forall$ and $\theta\_\exists$. The meaning is that

$$\begin{aligned} o_1\ \theta\_\forall\ o_2 \quad &:\Leftrightarrow \quad \forall_{v_1 \in o_1} \exists_{v_2 \in o_2} :\ v_1\ \theta\ v_2 \\ o_1\ \theta\_\exists\ o_2 \quad &:\Leftrightarrow \quad \exists_{v_1 \in o_1} \exists_{v_2 \in o_2} :\ v_1\ \theta\ v_2 \end{aligned}$$

Furthermore, there are operations on *instants* and *timeIntervals* that result in values of temporal types. It is possible to, e.g., perform intersection, union and difference on values of these types. "Metric" operations can be applied that calculate, e.g., the total duration of a *timeIntervals* value. Arithmetic operations allow to add or subtract from values of the temporal types, to shift a *timeInterval* value a given amount, etc.

The concept of a *history* serves as the basis for time-dependent entities. In particular, if such entities are spatial values, the resulting history represents a (discretely) spatio-temporal entity. According to Griffiths et al. (2001a), a history is a quadruple $H = \langle V, \theta, \gamma, \Sigma \rangle$, where V denotes the domain of values whose changes H records, θ is either *instants* or *timeIntervals*, γ is the granularity of θ, and Σ is a set of pairs, called *states*, of the form $\langle \tau, \sigma \rangle$, where τ is a Tripod timestamp and $\sigma \in V$ is a snapshot. The most important constraints on a history are: (1) a particular timestamp is associated with at most one snapshot; and (2) a particular snapshot is associated with at most one timestamp. The latter constraint implies so-called *coalescing* performed by the system. According to Jensen et al. (1998, p. 386), the coalesce operation is performed by packing as many value-equivalent entities as possible into a single value-equivalent one.

As an extension to ODMG ODL, the keyword "historical" is introduced to indicate that the temporal development of an attribute or an object is to be recorded. This keyword can be applied to each ODMG object model concept that is value assignable: object types, attributes, relationships, and the ODL collection object types. The authors refrained from taking an alternative approach of a *type constructor*. Such a type constructor must be applicable to itself such that

$$\text{historical}<\text{historical}<\text{set}<\text{historical}<\text{region}>>>>$$

becomes a valid expression, but the meaning of which is hard to conceive (Griffiths et al., 2001b, p. 15).

Being part of an interoperable object-oriented GIS-framework for atmospheric modelling, the geo-object-oriented database core GOODAC comprises two components: (1) the ODMG-based data model OOGDM and its temporal extension T/OOGQL; (2) the query language OOGQL (Bernard et al., 1998). OOGDM extends the ODMG data model by offering a class hierarchy that comprises $0d$ to $3d$ features, both field- and object-based. Hence, types are available for both representing point, curve, surface, and solid data, as well as $2d$ and $3d$ grids. Temporal extensions to the data model comprise valid and transaction time (Voigtmann et al., 1996). A timestamp is either a chronon or an interval on a linear and discrete time-line. T/OOGDM allows timestamping of simple types (byte, short, long, float, double, boolean, and string) or of relationships, whereas timestamping of object and collection types is not allowed. The keywords "chronon_timestamped", "interval_timestamped", and "timestamped" (shorthand for "interval_timestamped") are offered to the user of T/OOGDM to indicate recording of temporal development. Predicates are available for testing binary relationships among timestamps i, j: $i \ \theta \ j$, where $\theta \in \{\text{precedes, meets, overlaps, contains}\}$ (Becker et al., 1997).

Spatio-temporal entities can be represented by timestamping the geometry-attribute of a so-called *geo_object*. Alternatively, when spatial entities are represented by raster based data types, the attribute of such a raster can be timestamped, e.g., the height attribute of a digital elevation model.

Böhlen et al. (1998) present the spatio-temporal query and data definition language STSQL, which is an extension of SQL-92. This language adds features from both the temporal and the spatial domain, but neglects inherent spatio-temporal features and, as a result, does not address the issues of continuous change or compact storage of time-dependent spatial objects. On the other hand, a main concern of STSQL has been upward compatibility with existing SQL-92 based databases and applications, to ensure a simpler transition to spatio-temporal databases. Temporal and spatial aspects are treated uniformly in the sense that they make up possible *dimensions* of an STSQL-relation: a tuple in the model is allowed to be *timestamped* by one or more valid/transaction-time intervals or to be *spacestamped* by one or more geometric attributes. Adding dimensions to a relation facilitates the so-called *upward compatibility* and *snapshot reducibility* (for details see Böhlen et al., 1998). However, temporal and spatial attributes can also be added as ordinary attributes.

A model for discrete change along with operators has been developed by Worboys (1994). The well-known spatial representational model of simplicial complexes (Egenhofer et al., 1990, definitions 3.2.5 and 3.2.6) is extended by bitemporal elements. Here, a bitemporal element is defined as the union of a finite set of Cartesian products of intervals from the temporal domain. Hence, the model can capture both valid and transaction time; the temporal domain is linear and discrete. A simplex in this model is extended by timestamping with a bitemporal element yielding a pair (S, T), called *ST-simplices*, while an *ST-complex* C is a finite set of ST-simplices satisfying the following properties (Worboys, 1994, slightly altered):

- When projected to space, the set of ST-simplices form a simplicial complex.
- If the ST-simplices $s_1, s_2 \in C$ are projected to space, then the fact that the projection of s_1 is face of the projection of s_2 implies the bitemporal element of s_1 covers that of s_2.

Worboys (1994) defines algebraic operators on ST-complexes, among which are extensions of equality, subset, and boundary operators. Particularly interesting is the so-called β-product that combines two ST-complexes by: (1) defining a common refinement of the spatial projections of the input ST-complexes; and (2) associating each simplex of the refinement with the bitemporal element that results from the operator β on input bitemporal elements (Worboys, 1994, for details see). Specialisations of this operator are the ST-union, ST-intersection, and ST-difference.

Chen and Zaniolo (2000) define a data model, query language and an internal representation along algorithms, called SQLST. They address the issue of point-based vs. interval-based query languages. Their model is capable of representing discrete change and builds upon Worboys‘s model. The issue of representing continuous change was left for future work.

One class of application deals with a special case of time-dependent spatial objects. Of primary interest are objects that (continuously) change their *location*, while other spatial properties remain unchanged, for example shape. These applications range from transportation business, location-based-services (LBS), mobile telephony to military; hence, objects of interest are, for example, automobiles, mobile telephone users, or aircrafts. Since state-of-the-art commercial DBMS are not an adequate tool for the handling of this kind of spatio-temporal data (Wolfson, 2002; Brinkhoff, 1999), several research efforts have been initiated. Such work is relevant to the topic of this thesis, since abstracting from the spatial extent also *natural objects*—like clouds or glaciers—can be handled by moving objects database systems. By offering query evaluation involving object *trajectories*, such systems can prove useful also in geoscientific applications.

DOMINO is among the moving objects database systems (Sistla et al., 1997; Wolfson et al., 1999) with focus on *tracking* moving objects (Wolfson et al., 1998). Central to this system is its *moving objects spatio-temporal data model* (MOST) that is based on a relational data model and comprises also spatial types (points, lines, and regions in two-dimensional space). Furthermore, MOST incorporates the concept of a dynamic attribute (Sistla et al., 1998). Representing functions of time, such attributes change without an explicit database update, and they can be used to model continuously changing values, for example, temperatures, but also object positions. Hence, movement in two-dimensional space is represented by two dynamic attributes that together make up a motion vector of the object. A dynamic attribute A is a three tuple

$$(A.value, A.updatetime, A.function)$$

comprising three sub-attributes: (1) $A.value$, which is equal to A's value at the time of last update; (2) $A.updatetime$, the time of last update; and (3) $A.function$, the function from time into the domain of A. The value of A is defined as $A.value$ for time $t = A.updatetime$ and $A.value + A.function(t)$, otherwise. In a running system, dynamic attributes are intended to be updated periodically by the corresponding real world entities, for example using GPS and communication over a wireless network. A slightly different model with six sub-attributes for trajectories along a network has been presented by Wolfson et al. (1999). The query language of MOST is FTL (future temporal logic) that extends a DBMS query language by spatio-temporal predicates and temporal operators. The former comprise distance- and intersection-based predicates between moving objects or spatial objects. The latter comprise two basic operators: *Until* and *Nexttime*. Further interesting aspects of DOMINO are the handling of uncertainty (Sistla et al., 1998) and the indexing of moving objects (Tayeb et al., 1998). However, since past states of an object's trajectory are not explicitly represented, the system is restricted to queries pertaining to the future.

Grumbach et al. (1998) show how to utilise the so-called constraint database approach (Kuper et al., 2000) to represent and query spatio-temporal objects. However, the model is limited to discrete change of spatial data.

Related to the approach based on constraints is the work of Yeh and de Cambray (1995) and Yeh and Feautrier (1998). The authors present a model for multidimensional data that is allowed to be a function of a variable. In particular, spatial data can thus be made a function of time. The authors distinguish an external and an internal level of modelling. In the former, a spatio-temporal object appears as a sequence $(v, t, c)_i$, where v is a spatial value valid for snapshot t and c is a so-called *behavioural function* that specifies spatial values between t_i and t_{i+1}. In the latter, a spatio-temporal object is represented as a collection of convex objects that are the result of a *sweeping* operation: if the spatial values in the sequence have n dimensions, then the sweeping operator constructs an $n+1$-dimensional object by applying the behavioural function for each time-step of the sequence. This results in a set of polyhedra, which the authors define to be convex objects. Each polyhedron is then stored internally either as a conjunction of inequalities (a convex object is the intersection of half-spaces) or as a set of points (the convex hull of which equals the polyhedron). Non-convex geometries are stored in operator trees, similar to constructive solid geometry. The benefit of this representation is that the temporal versions of the operations *intersection, union, difference* can be realised by their spatial versions operating on the internal representation. However, other important operations like the Euclidean distance remain a challenge. Furthermore, by representing spatio-temporal objects through $n + 1$-dimensional polyhedra, movement is restricted since polyhedra are defined to have non-curved (i.e. linear) bounding faces.

Chomicki and Revesz (1999b) present a two-dimensional data model that is based on vertex movement, which they call *parametric 2-spaghetti data model*. It generalises Worboys's model (Worboys, 1994) by allowing vertex positions to be linear functions of time. The negative result is that the intersection of two parametric 2-spaghetti relations cannot be always represented as a parametric 2-spaghetti relation.

Based on the relational model, Cai et al. (2000) present the so-called parametric rectangles spatio-temporal data model. In this model, a relation can draw zero or one of its attributes from the new domain $\mathcal{P}$, the set of all parametric rectangles. In particular, an element of this domain is a tuple

$$(x_1^{[}, x_1^{]}, \ldots, x_n^{[}, x_n^{]}, \text{from}, \text{to})$$

where each $x_i^{[}$ and $x_i^{]}$ is a polynomial function from $f : \mathbb{R}_{|[\text{from},\text{to}]} \rightarrow \mathbb{R}$. The meaning is such that for each dimension i a time-dependent interval is given by the functions $x_i^{[}$ (lower boundary) and $x_i^{]}$ (upper boundary). Then, for each $t \in [\text{from}, \text{to}]$ an n-dimensional, axis-aligned rectangle is defined by the cross product of the n intervals $[x_1^{[}(t), x_1^{]}(t)] \times \ldots \times [x_n^{[}(t), x_n^{]}(t)]$.

In this model, a spatio-temporal object can be represented by a set of parametric rectangles. The query language of this model is an extended version of relational algebra. The operators projection, intersection, union, difference, and complement are defined for relations with parametric rectangle attribute, whereas selection retains its usual semantic. Furthermore, the operators *block*, *collide*, *deflect*, *scale*, and *temporal* operators are defined. To sum up, the particular strengths of the model are its dimension-independence, the facility to approximate complex shapes of spatio-temporal objects including also non-linear motion patterns, and the closure of the algebraic operators defined for the model.

However, in the parametric rectangles data model, complex spatio-temporal objects are not first-class-citizens: they must be represented by several tuples. This holds for the following situations: (1) for objects that have a more complex shape than a rectangle; (2) for a rectangle that changes direction or speed in one of its interval end points, since the corresponding temporal function is then invalidated and a new parametric rectangle tuple must be generated to capture that change in movement. As a result, one spatio-temporal object must be stored either in a separate relation or it must be "intermixed" with other objects. Several difficulties arise from this design. First, resulting from the fact that one object "spreads" across several tuples, no trivial way exists to index a set of spatio-temporal objects. Hence, the parametric rectangle data model has the problem of *indexing* spatio-temporal objects. Second, for the same reason it is also unclear how a spatio-temporal object can participate in the predicate of a *select*-operation, for instance in the predicate $\text{distance}(o_1, o_2) \leq 50$.

These difficulties can be avoided when allowing spatio-temporal objects to be first-class-citizens, for instance by one level of nesting of a relation. Another possibility lies in an *abstract data type approach*. Assuming the relational data model as the underlying basis, a single spatio-temporal object can then be represented as the attribute of a single tuple.

An approach based on abstract data types has been described by Güting et al. (2000). The authors define an abstract model for continuous change of data with the focus on changing 2D spatial data. Starting with a set of basic (including spatial) data types, a type constructor "moving" is defined that, if applied to one of the basic data types, constructs a new type that represents a mapping from time into the domain of the argument to "moving". Hence, "moving<region>" would construct a time-dependent region data type, whereas "moving<real>" would construct a data type that maps time instants to real-values. The spatial data types extended in such a way comprise two-dimensional points, curves (not necessarily simple), and regions (allowed to have holes and may consist of several connected components). The description is complemented by operations defined on these data types. An interesting aspect is that of *lifting* of operations. This technique extends the signature of operations on the basic data types and their semantics to be applicable to data types constructed by the "moving" type constructor. The signatures are extended such that the return type T as well as any number of arguments are transformed to "moving$< T >$", yielding a set of operation signatures.

A first step towards the implementation of the model by Güting et al. (2000) has been described by Forlizzi et al. (2000). The authors propose a data model that enables a finite representation of the spatio-temporal data types mentioned in the previous work. Central to the approach is a *sliced representation* of spatio-temporal objects. According to the sliced representation, a time-dependent object is given by a finite sequence of snapshots, while the object can be interpolated for instants in between. Movement and change of extended spatial objects (i.e., those other than points) is based on a restricted form of point movement: between two slices the movement of those points, which are connected to form a segment, must be within a plane in (x, y, t)-space. Furthermore, it is briefly described how the specification of the data model can be mapped onto the secondary storage structures of the Secondo extensible DBMS (Dieker and Güting, 2000). Finally, the realisation of two operations are described: a snapshot query and the binary predicate "inside". The latter is restricted for the parameters moving point inside moving region.

An interesting framework for specifying spatio-temporal objects has been reported by Chomicki and Revesz (1999b). This framework is based on the conception of a spatio-temporal object as the *transformation* of a spatial reference object over time. Hence, the authors define an *atomic* geometric object to be a quadruple (V, v, I, f), where (Chomicki and Revesz, 1999b)

- V is the reference spatial object,
- v is a reference time instant,
- I is a subset of $\mathbb{R}$, the temporal domain of the framework,
- f is the transformation function.

The semantics of such an object[1] is given by the set of pairs (V', i), resulting from the application of $f(V, i - v)$ for each $i \in I$ resulting in a new spatial object V'. Further, the authors define a molecular geometric object as a finite set of atomic geometric objects whose time domains are disjoint. Assuming the temporal domain as intervals over the reals, both the spatial domain as well as the domain of transformation functions may vary. One obtains then different classes of spatio-temporal objects. Chomicki and Revesz (1999b) suggest several such classes that are then investigated concerning closure under set-theoretic operations *intersection*, *union*, and *difference*. For instance, it is claimed that the class of axis-aligned rectangles (spatial domain) with linear scaling (transformation function) is closed under the aforementioned operations, whereas the class of convex polygons with linear scaling is not closed under intersection or difference (Chomicki and Revesz, 1999b). Finally, choosing the *identity* as the transformation function the framework is also capable of representing discrete change.

The field of qualitative spatio-temporal reasoning is an ongoing field of research (see also Galton (1995); Muller (1998); Erwig and Schneider (1999); Hazarika and Cohn (2001); Erwig and Schneider (2002)). The implementation aspect remains still a challenge.

[1]Slightly revised definition

Chapter 3

A Representational Model for Spatio-Temporal Data

The representational model for spatio-temporal data plays a central role in spatio-temporal database systems: it precisely defines the entities that can be represented (a topic covered in this chapter), it states requirements for the internal representation (covered in chapters 4 and 5), and it forms the basis for operations and their realisation (covered in chapter 6). At the same time, current systems are not well equipped with spatio-temporal data handling capabilities and a consensus representational model has not yet emerged (see also chapter 2).

In this chapter, a new model is presented as the basis for representing, storing, and querying spatio-temporal data. Originating in previous work within the GeoToolKit projects, several aspects of the representational spatio-temporal model are not new. Instead, the new model *consolidates* main characteristics of the previous work, and one result of this chapter is the formalisation of the new model. Furthermore, this chapter investigates the *temporal* domain in detail. The findings are the requirements for temporal data types that are to complement the data types for time-dependent geometries from previous projects. An implication of these findings is the fact that the prevalent representations of time in temporal database models do not comply with the requirements of the spatio-temporal (geometric) settings. Finally, this chapter discusses capabilities and limitations of the new model. Here, it is first emphasised that the model is purely representational. Furthermore, since the objects that the model represents are time-dependent point sets in three-dimensional space[1], it becomes worth investigating the closure of set-based operations like union or intersection. One result of the chapter states its non-closure under these set-based operations.

The remainder of the chapter is organised as follows. The next section reviews GeoToolKit projects and their model for spatio-temporal data. Then the main characteristics are put together by introducing the new model for spatio-temporal data. The chapter closes with the investigation of the model‘s capabilities and limitations.

[1]More precisely, these objects are *approximations* of time-dependent point sets in three-dimensional space.

3.1 Influences from Previous Projects

Previously, joint projects with partners from computer science and geology, both located at Bonn University, dealt with temporal aspects of the database management for geoscientific phenomena. These efforts resulted in two different database models for the representation of continuously changing spatial data: a development from the management of different timesteps to separating vertex movements for triangular meshes. Bringing characteristics together, these models evolve then into a new model of temporal simplices and temporal simplicial complexes as described in the next section.

3.1.1 Balanced Restoration of Structural Basin Evolution

Concerning the management of time-dependent geo-objects, several requirements emerged during a joint project together with the Geological Institute at Bonn University[2]. In this section, the project and its representational model for spatio-temporal data on the database system side are described briefly. Further information about this project can be found in Alms et al. (1998) or Breunig (2001).

First ideas for the management for temporal spatial data have been implemented in GeoToolKit to support the balanced restoration of the structural basin evolution for the Lower Rhine Basin (Balovnev et al., 1999). The bases for the modelling were interpreted lithostratigraphic sections with horizontal scale 1:10000 and vertical scale 1:2000. Parallel sections run in the SW-NE direction and cover a distance of 1.5 to 4.5 km. The computer supported process of geological reconstruction was made with help of GeoDeform (Alms et al., 1998). This application is well suited to demonstrate the spatio-temporal modelling and management of geo-objects. By coupling with GeoToolKit, GeoDeform benefits from database management including interactive selection of spatio-temporal objects. Therefore, GeoToolKit has additionally been designed according to requirements of this application. The management of temporal strata and faults was enabled with the addition of a further attribute *time* in the corresponding objects (Balovnev and Breunig, 1997). The respective attribute value shows at which time or time interval the object existed in geological reality. Though developed for a specific application, the proposed model for spatio-temporal data handling is assessed as transferable to a wider range of geo-applications (Balovnev and Breunig, 1997).

On the database system side, the representational model for spatio-temporal data has been designed in such a way that it achieves compatibility to the graphical programming environment GRAPE[3]. The system GRAPE is an object-oriented, interactive visualisation package that offers, among others, an extension of the key-frame interpolation technique for adaptive triangulations (Polthier and Rumpf, 1994). It has been chosen as the basis of GeoDeform (Alms et al., 1998). Importantly, the extended key-frame interpolation technique leads to a time-dependent surface type incorporated into the database environment of Geo-

[2]Projects within the Collaborative Research Centre 350: *Interaction between and Modelling of Continental Geo-Systems*, founded by the German science foundation (Deutsche Forschungsgemeinschaft, DFG); subprojects C4 "Quantitative Modelling of the Tectonic and Sedimentary Development of the Lower Rhine Basin" and D4 "Object-Oriented Geo-Information System".

[3]Developed at the collaborative research centre 256 "Nonlinear Partial Differential Equations" at the University of Bonn.

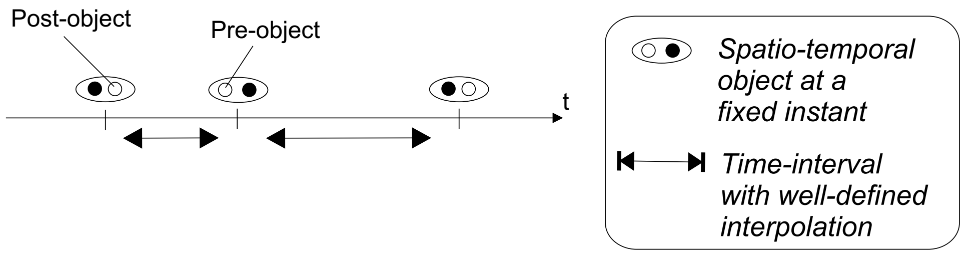

Post-object
Pre-object
t
Spatio-temporal object at a fixed instant
Time-interval with well-defined interpolation

Figure 3.1: Schematic view on the development of a spatio-temporal object in the GRAPE model. Each snapshot consists of a pre- and a post-object that have the same geometry, but are allowed to have different discretisation of this geometry. However, post-object and pre-object of two consecutive snapshots have the same discretisation, enabling interpolation of the geometry for instants between snapshots.

ToolKit. The extended key-frame technique can be described as follows (see also Polthier and Rumpf, 1994). A schematic view is shown in figure 3.1. Central to the model is a time-dependent *surface* type. Such surfaces are composed of *triangles* that together form a *triangular mesh.* A time-dependent surface is represented by a sequence of so-called snapshots at discrete time instants, each of which contains *two* (static) triangular meshes: a so-called pre-object and a so-called post-object associated with an instant in time. Hence, a spatio-temporal surface can be seen as an ordered collection of purely spatial surfaces, each surface being a snapshot of the spatio-temporal surface. Together with the requirement that the discretisation of the post-object at instant t_i and the discretisation of the pre-object at the consecutive instant t_{i+1} is the same, *interpolation* of the time-dependent surface can be defined. Since there is a one-to-one correspondence between the triangles of the post-object at time t_i and the pre-object at time t_{i+1} (the meshes are homeomorphic), an object can be *interpolated* for every instant between t_i and t_{i+1}. Furthermore, the discretisation of the surface is allowed to change at an instant under the requirement that both pre- and post-object describe the same geometry.

The database implementation of the model, however, can result in a high storage demand, because the data are stored multiply. This is due to the time-stamping that is performed on the level of the compound object, instead of time-stamping on the level of the object‘s elements. Experience showed that only in rare cases the discretisation of a surface changes completely from one timestep to the next, while it is expected that many mesh elements remain part of the surface discretisation, ranging over many snapshots. As a result, these mesh elements occur in all of these snapshots, which is obviously redundant. Time-stamping on the level of a surface‘s elements can overcome this problem: a fact that is exploited in the next section when defining the new representational model.

A first approach to cope with the storage space problem has been based on *versions.* If, for example, only a small part of a large object changes in time, it is “wasteful” to reproduce the stable part of the object in each snapshot. Here, the *version-based* approach stores full versions at key-frames, while storing Δ-versions (*differences* between snapshots) in between. A description of such an approach for the model has been given by Shumilov and Siebeck (2001).

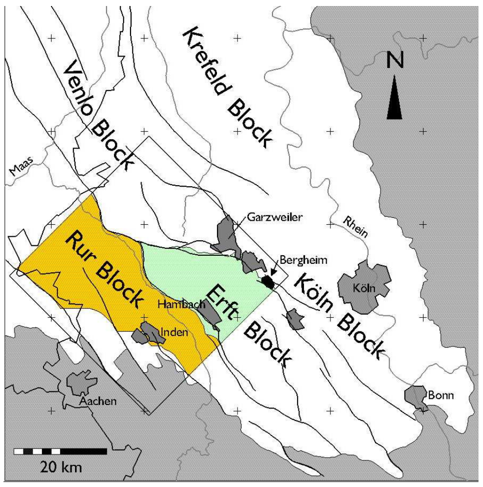


Figure 3.2: Position of the Bergheim mine in the Lower Rhine Embayment.

3.1.2 Kinematic Modelling of a Small Faulted Domain

During a subsequent cooperation, a new model for time-dependent surfaces has been developed by separating vertex movement from mesh topology; however, topology has been assumed to be static as opposed to the previous model. The project and its representational model for spatio-temporal data are described briefly. Further information about the project have been given by Thomsen and Siehl (2000).

The temporal model of the (extended) GeoToolKit has been tested with a geological model consisting of stratigraphic and fault surfaces extended by a time attribute. The animated surfaces of that model were visualised in the geological modelling software system Gocad (Mallet, 1992) and stored using GeoToolKit for spatio-temporal database querying.

One of the objectives of the geological model was to examine the balanced restoration of structural Rhine Basin evolution (Thomsen and Siehl, 2000). This geological process started in the geological period called tertiary. The exact geological movements, however, are unknown. Therefore, different possible variants for the time sequences have to be examined by the geologists. The region under consideration covers the area of an open pit lignite mine "Tagebau Bergheim" and comprises an area of about 2km×2km up to a depth of 0.5km. The Bergheim mine was operated by Rheinbraun AG, Köln, since 1984 with an annual output of 15.7 Mill. tons (1996/97) of lignite. The originally static 3D-model on the basis of approx. 20 digitised profile sections of Rheinbraun AG was prepared for the movements. The 3D geometric model consists of a boundary representation of geological blocks sliding against each other at fault boundary surfaces. For the reconstruction of the movements in time Rouby‘s method of map plane restoration was used (Rouby et al., 1996). The system Gocad served for static 3D modelling and visualisation, thereby using the Weiler model (Weiler, 1985, 1986) that is implemented in this system and assures a consistent 3D model; however, this model cannot be made time-dependent. Therefore, a representation extended by time has been stored within GeoToolKit, complemented by sufficient information to reconstruct a (valid) Weiler model for every instant in time to enable import into Gocad.

The database model for spatio-temporal data serving this application can be described as follows. Central to the model is a time-dependent *surface* type. Such surfaces are composed of *mesh elements* (triangles for every instant in time) that together form a *triangular mesh*. For the whole lifetime of a surface the set of its constituent mesh elements remains fixed. Hence, the discretisation of a surface is constant for its whole lifetime. Movement and deformation are representable through movement of the vertices of the mesh elements; like the previous model, every instant is associated to a surface. Each vertex is allowed to move on a piecewise linear path with piecewise constant velocity. This aspect of the model will carry over to the new model presented in the next section.

An important aspect inherent to the model is the separation of vertex movement from mesh elements. Each triangle in the triangular mesh holds three *references* to a (temporal) vertex: a list of point snapshots representing the movement. This approach has the advantage of compact storage, since triangles that share a common vertex do not duplicate its trajectory; instead, they share a reference to a separate object: the time-dependent, moving vertex. Moreover, the approach of separating vertices from mesh elements has advantages that also hold in the pure-spatial setting. First, redundancy is avoided and therefore also update anomalies. Second, it allows for constructing several meshes from the same set of vertices. For all these reasons, the separation of point location and meshing will carry over to the new model.

3.1.3 Summary

Within the GeoToolKit projects (Balovnev et al., 2003) two different models for the management of spatio-temporal data have been developed and tested within applications.

The particular strengths of the first model have been the ability to model discontinuities and to change the discretisation of objects in the course of time. Problematic in such models is storage space, since time-stamping is on the complex object level such that different timesteps necessarily repeat mesh elements. An approach to alleviate these problems has been based on maintaining versions. The particular strengths of the second model have been the separation of vertex movement from mesh elements and thereby facilitating a compact representation. In contrast to the first model, it did not allow for a change in discretisation.

These models along with their advantages have not been formalised and they have not been unified into a common model. Additionally, temporal data types were missing. This consolidation is subject of the next section. Furthermore, spatio-temporal operations on this data have been limited to a temporal range query and, as a special case, a snapshot query on time-dependent geometry, capable of interpolation between snapshots. Further spatio-temporal operations are discussed in subsequent chapters.

3.2 Spatio-Temporal Representational Model

A spatio-temporal representational model comprises three parts. First, there is the underlying *temporal* structure, which is given by definition of a temporal domain and its operations. The elements of this domain are used to denote the lifetime of objects or the time when facts are considered true in the database. Second, there is the spatial domain, which is extended to capture also *time-varying* spatial data. Third, there is the spatio-temporal domain. Here, temporal and spatial aspects are put together in that spatio-temporal entities represent certain *functions* from the temporal domain into the spatial domain.

Compared to section 3.1, the particular new work here is as follows. First, the pure-temporal domain is investigated to infer requirements for temporal data types. Such types have been absent from work within the GeoToolKit-projects. Second, main characteristics from both previous approaches are put together by defining the novel concepts of a *temporal simplex* and a *temporal complex.* The resulting model facilitates the representation of topological change in the course of time, as well as a separation of a mesh and the location of its vertices. In summary, the representational model comprises not only a separate temporal and spatial domain as does most related work (see chapter 2), but also a combined spatio-temporal domain. The remainder of this section discusses the three constituent domains.

3.2.1 Temporal Domain

This section investigates the temporal domain for the spatio-temporal representational model and infers requirements for temporal data types. In particular, the choices for three different concepts are explored: (1) the temporal domain itself and the concept of an instant, which is an element of this domain and represents a (durationless) point in time; (2) a definition of admissible subsets of the temporal domain that specify the lifetime of objects, resulting in so-called temporal elements, which are disjoint sets of temporal intervals; a further result is that intervals must be defined as connected subsets of the temporal domain and may be open on one or both sides and may have zero duration; (3) operations on the mentioned entities. The centre of interest lies in spatio-temporal data representation that guides the definitions of these concepts.

Given that the temporal domain consists of temporal points, also termed *instants*, several reasonable options for the structural aspects of time have been discussed in research (see, for example, Snodgrass, 1992). In particular, these options have connections to the order relationship among these instants ("before" or "<"). Of interest are: (1) a *linear* conception of time, open on both sides (*past* and *future*); this conception yields a total order of instants; (2) a *circular* conception of time, suitable, for instance, for periodic data; (3) a *parallel* conception of time consisting of more than one linear time-line; (4) a *directed, acyclic graph* conception of time, also called *branching time*, yielding a partial order relationship. Although all of these types of time are worth investigating in spatio-temporal database systems (Raza and Kainz, 1999), this thesis concentrates upon the linear conception of time for the following reasons: (1) The linear conception of time is prevalent in temporal databases (Snodgrass, 1992); (2) it is well suited for measurements/observations that follow also a linear pattern; (3) finally, it is also well suited for simulations, which usually assume a temporal variable ranging over the set of real numbers. Having decided on the linearity of time, the next question is whether time

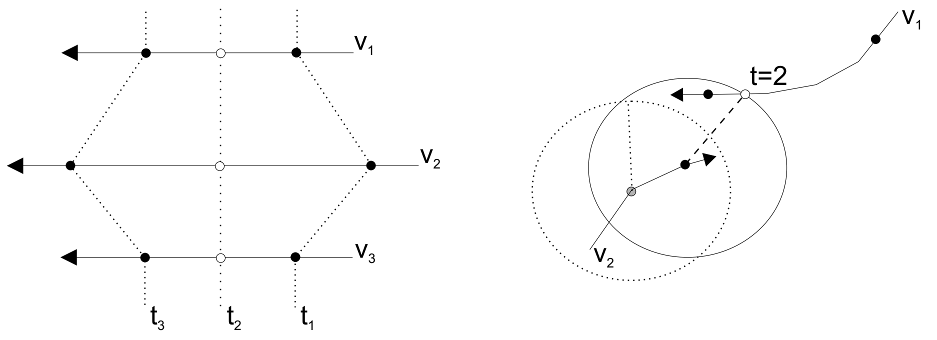


Figure 3.3: Examples for events without duration. Left: The time when three moving vertices are colinear. Right: The time when moving vertex v_1 has distance d to moving vertex v_2.

is discrete or dense; however, since in a discrete time domain it is problematic to describe *continuous* change of a variable, time is assumed to be dense. To sum up, the temporal domain is isomorphic to the set of real numbers.

A time-dependent value from a given attribute domain $\mathcal{D}$ can be seen as a function from any subset of the temporal domain $\mathbb{T}$ into the attribute domain $f : \mathbb{T} \to \mathcal{D}$. For tractability, however, the admissible subsets of $\mathbb{T}$, which may specify the domain of f, must be restricted. It is common practice to restrict the domain of such a function f to *intervals* of $\mathbb{T}$ or finite *sets of intervals* of $\mathbb{T}$. Alternatively, one can describe infinite sets of intervals, for instance, by periodic intervals. In their spatio-temporal model, Cai et al. (2000) represent such sets of intervals by a triplet $< \mathit{from}, \mathit{to}, \mathit{period} >$ to denote the (infinite) set of intervals $\{< \mathit{from} + k \cdot \mathit{period}, \mathit{to} + k \cdot \mathit{period} >\}$. Here, however, periodic intervals are not considered. Then, having an intuitive notion of intervals (precisely defined below), it must be examined how "general" they need to be: (1) it is in question if intervals can be assumed to be closed, to be right-open, or if rather both open and closed intervals are needed; (2) it is in question if intervals are allowed to have zero duration. Obviously, the different choices influence upon interval representation, for instance, space demands for storing a single interval. Indeed, these questions have been discussed in research, of course, independently from space demands. For example, in the popular system of Allen (Allen, 1983), instants are used to construct intervals, but instants are not the primitive objects in his theory and every interval has a non-zero duration. This means that it is impossible to represent facts that are valid at a single instant only or to represent facts that are not valid at the boundary of an interval. One of Allen's motivation for this restriction has been the physical argument that "there seems to be strong intuition that, given an event, we can always 'turn up magnification' and look at its structure. [...] it appears that we can always decompose times into subparts." (Allen, 1983, p. 834). Furthermore, he argued that in some situations counterintuitive results appear. These are also known as the *dividing instant problem* (see also Galton, 1995). For instance, the state of a time-dependent variable *light-is-on* changes when light is turned on or off. Given that light is turned on at time t_2, *light-is-on* is false during $< t_1, t_2 >$ and true during $< t_2, t_3 >$. If one allows instants as primitives, one must specify the value of *light-is-on* at instant t_2, what seems problematic in any case. Hence, Allen rejects instants as primitive objects. While the dividing instant problem can be circumvented by a different modelling approach (Galton, 1995), there is also strong evidence for the necessity to model events without duration, especially in the geometric setting of spatio-temporal databases. Figure 3.3 shows some motivating examples. If instants were not supported, problems would arise when applying operations that result in spatio-temporal objects or timestamps. For instance, the query

When was the distance between spatio-temporal objects v_1 and v_2 equal to 100?

will result in an *instant*, assuming no degenerate cases (the case of movement of v_1 along the boundary of the circle around v_2 with radius 100 for a duration bigger than zero, or the case that the distance never reaches 100). For a similar reason, intervals must meet a further requirement. It must be possible to represent not only closed, but also *open* and *semi-open* intervals. This becomes obvious when replacing “equal to 100” with “less than 100” in the above example. As a result, instants must be present in spatio-temporal databases in the sense that intervals are allowed to have zero duration and are allowed to be open on one or both sides. Consequently, the extra space demand for interval representation[4] is an immanent aspect.

Furthermore, there must be a means to represent that a fact is never true. For instance, if in the above example object o does not reach a distance of 100 to point p then the fact *“Object o has distance 100 to point p”* is never true and the result of the above query must reflect that. Consequently, a special interval is introduced:

$$(\mathbf{undef}, \mathbf{undef}) = \emptyset$$

Here, **undef** augments the temporal domain as a special symbol and is used to construct the interval $\emptyset$.

A further requirement stems from the handling of pure-spatial data and the question of its temporal validity. Spatial data is not related to time and when it is injected into the spatio-temporal domain, the only options are: (1) it is assumed to be never valid such that spatial data implicitly carries the (special) interval $\emptyset$; and (2) it is assumed to be always valid. The first option implies a strict separation between spatial and spatio-temporal data, contradicting the goal to integrate both kinds of data. Therefore, it is assumed that spatial data is valid at all instants. The special symbols $-\infty$ and ∞ are introduced into the time domain for this purpose. Their meaning is such that every instant is bigger than $-\infty$ and less than ∞. Then, spatial data can implicitly carry the interval of validity $(-\infty, \infty)$. The following definitions sum up the arguments given so far.

3.2.1. Definition. (*time domain, “<”*) The *time domain* $\mathbb{T}$, interpreted as being isomorphic to the set of real numbers, is the structure:

$$\mathbb{T} = (\mathbb{R} \cup \{-\infty, \infty, \mathbf{undef}\}, <)$$

[4]Not only are intervals specified by their left and right boundary, but also by the necessary information of open- and closeness.

The relation "<" is extended as follows. Let $p_1, p_2 \in \mathbb{T}$:

$$p_1 < p_2 = \begin{cases} p_1 < p_2 & \text{if } p_1 \in \mathbb{R} \text{ and } p_2 \in \mathbb{R} \\ \text{false} & \text{if } p_1 = \infty, p_2 \neq \textbf{undef} \text{ or } p_2 = -\infty, p_1 \neq \textbf{undef} \\ \text{true} & \text{if } p_1 \in \mathbb{R} \cup \{-\infty\} \text{ and } p_2 \in \mathbb{R} \cup \{\infty\} \\ \textit{undefined} & \text{if } p_1 = \textbf{undef} \text{ or } p_2 = \textbf{undef} \end{cases}$$

□

Special symbols $-\infty$, ∞, and **undef** are available. Relation "<" is defined on any two elements in $\mathbb{T} \setminus \{\textbf{undef}\}$, but undefined for **undef**, therefore the extension of "<" is a partial order. Intervals as described above are the primitives used to model the lifetime of objects. The relation $p \leq q$ is defined as $p < q \vee p = q$.

3.2.2. DEFINITION. (*interval*) Let p_l, p_u be two instants in $\mathbb{T} \setminus \{\textbf{undef}\}$ such that $p_l < p_u$. An *open interval* takes one of the following forms:

- A *left-open interval* is defined as $(p_l, p_u] := \{t \in \mathbb{T} \mid p_l < t \leq p_u\}$
- A *right-open interval* is defined as $[p_l, p_u) := \{t \in \mathbb{T} \mid p_l \leq t < p_u\}$
- An *open interval* is defined as $(p_l, p_u) := \{t \in \mathbb{T} \mid p_l < t < p_u\}$

Let p_l, p_u be two instants in $\mathbb{T} \setminus \{\textbf{undef}\}$ such that $p_l \leq p_u$. A *closed interval* is defined as follows: $[p_l, p_u] := \{t \in \mathbb{T} \mid p_l \leq t \leq p_u\}$. The *lower boundary* of an interval i is defined as $\inf i$ and denoted by i^-, the *upper boundary* is defined as $\sup i$ and denoted by i^+. The *boundary* of i, denoted by ∂i, is defined as $\{i^-, i^+\}$. The *closure* of i is defined as the closed interval $[i^-, i^+]$. Finally, the *sets of intervals* $\mathbb{I}$ is defined as the union of the sets of open and closed intervals and the set $\{\emptyset\}$. □

Note that it is possible to form a degenerate interval $[t, t]$. In the sequel, such intervals are used to represent that a fact is valid at a single instant. Differently, intervals like $[100, 10]$ do not conform with definition 3.2.2.

Boolean operations of the kind $f : \mathbb{I} \times \mathbb{I} \to \{true, false\}$ are useful in query conditions, as in the temporal selection *Retrieve all objects whose valid time intersect the interval* $[100, 200]$. Such operations have connections to *qualitative reasoning* about intervals, and the approach usually taken is that of Allen (1983). He identified a minimum set of possible relationships among intervals. However, as noted above, Allen does not allow instants as primitive objects. As a consequence, these relationships must be extended to account for instants and (semi-) open intervals. Ma and Knight (1994) have given a solution on a logical level, presenting a unifying theory including both intervals and points. The operational form derived in this thesis is topic of the next chapter.

The *intersection* operation $\cap : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ is defined in the usual set-based way; hence, it is also defined correctly for the special interval $\emptyset$. Except for *intersection*, intervals are not closed under set-based operations and are therefore not adequate to model natural language queries involving "and", "or", and "not" (see also Gadia and Yeung, 1988). Furthermore, the result of *when*-operations (*When has object o met a given predicate?*, see also chapter 6) is, in general, a *set* of intervals. This leads to the concept of a temporal element.

3.2.3. Definition. (*temporal element*) A *temporal element* is any finite subset of $\mathbb{I}$. A temporal element e is said to be *minimal* if for all pairs of distinct intervals $i, i' \in e$ the condition $i \cap i' = \emptyset$ holds. The set of temporal elements is denoted by $\mathbb{E}$.

Temporal elements are closed under set-based operations. It is worth noting that for every temporal element e in $\mathbb{E}$ there is a uniquely defined minimal temporal element $\min(e)$ such that $\bigcup_{i \in e} i = \bigcup_{i' \in \min(e)} i'$. Hence, minimal temporal elements correspond to equivalence classes of $\mathbb{E}$. Equality of temporal elements e_1, e_2 can then be reduced to membership to the same equivalence class ($e_1 = e_2 \Leftrightarrow \min(e_1) = \min(e_2)$). Therefore, in the sequel the term "temporal element" usually refers to "minimal temporal element". Temporal elements are used as timestamps, that is specifications of lifetimes and thereby domains of functions from time into space.

3.2.4. Definition. The *boundary* of a temporal element e is defined as the set of boundary instants of each interval in $\min(e)$. The *closure* of e, denoted by closure(e), is defined as the temporal elements that consists of the closures of e's intervals. The *hull* of e, denoted by hull(e) is the smallest, closed interval that contains all of e's intervals. More formally, these concepts are defined as follows:

$$\begin{aligned}
\partial e &:= \bigcup_{i \in \min(e)} \partial i \\
\text{closure}(e) &:= \bigcup_{i \in e} \text{closure}(i) \\
\text{hull}(e) &:= [\min_{i \in e}\{i^-\}, \max_{i \in e}\{i^+\}] \\
e^+ &:= \max_{i \in e}\{i^+\} \\
e^- &:= \min_{i \in e}\{i^-\}
\end{aligned}$$

□

To sum up, the temporal domain is a structure isomorphic to the set of real numbers, extended by the symbols $-\infty, \infty$ and **undef** for the reasons mentioned above. One element of this domain is called an *instant.* It is possible to construct degenerate intervals of the form $[t, t]$. Furthermore, there is the special temporal interval $\emptyset$ to denote that the lifetime of an object is empty. A temporal interval may be open on one or both sides. The set of temporal intervals $\mathbb{I}$ is the building block for timestamps, or *temporal elements*, subsumed by the set $\mathbb{E}$. Elements of $\mathbb{E}$ are used subsequently to define the lifetime of spatio-temporal objects.

3.2.2 Spatial Domain

In this thesis, geo-spatial objects are assumed to be embedded in a *continuous* space under the Euclidean metric. Many spatial data structures have been proposed in literature to represent spatial objects. Such representations typically store *topological* information of a configuration of spatial objects. For instance, the Node-Arc-Area or Doubly-Connected-Edge-List data structures in two-dimensional space are prominent examples (see also Worboys, 1995). In three-dimensional space the Weiler data structure (Weiler, 1985, 1986) or the g-map data structure (Lienhardt, 1994) of the Gocad geological modelling system (Mallet, 1992, 2002), boundary representation (B-REP, Mäntylä, 1988), constructive solid geometry are used to model spatial objects while representing also topological information to a varying degree. It is worth noting that current commercial database systems as back-ends of geographical information systems are limited to two-dimensional data without direct support for topological information.

The *simplicial complex model* is among the topological models for spatial data (Egenhofer et al., 1990; Breunig, 1996). It has been chosen for the GeoToolKit projects (Balovnev et al., 1997) and it is also adopted as the spatial domain for this thesis (Shumilov and Siebeck, 2001). The central concepts are primitive objects, called *simplices*, which are the building block for complex objects, called *simplicial complexes*. These complexes are sets of simplices conforming to a topological constraint: the intersection of two simplices must also be contained in the complex. This model proved adequate for applications of the GeoToolKit projects (Balovnev et al., 1998), while other application domains can also be supported, for instance terrain modelling (Van Kreveld, 1997).

While these concepts are dimension-independent, for the scope of this thesis the number of dimensions will be fixed to *three* spatial dimensions. The reason is the capability of *integration* of datasets ranging from different disciplines like geography, geology, soil science, or meteorology, the underlying space for the representational model is assumed to be three-dimensional. Examples for the following definitions of simplices and simplicial complexes are given in figure 3.4.

3.2.5. Definition. (*d-simplex*) Let P denote a set of $d+1$ affine independent points. The convex hull of P, $conv(P)$, is called a *d-dimensional simplex* (d-simplex for short). The elements in P are called the *vertices* of a simplex s and are denoted by $s^{(i)}$, $0 \leq i \leq d$. The dimension of s is denoted by $\dim(s)$. The convex hull $conv(P')$ of each subset $P' \subseteq P$ is called a *face* of s, denoted by $f_{\{i_0,\ldots,i_j\}}(s) = conv(\{s^{(i_0)},\ldots,s^{(i_j)}\})$ with $i_0,\ldots,i_j \in \{0,\ldots,d\}$ pairwise disjoint, $j \leq d$. The set of points in $\mathbb{R}^3$ that form a simplex s, denoted by $points(s)$, is given by

$$points(s) = \left\{ p \;\middle|\; p = \sum\nolimits_{i=0}^{d} \lambda_i s^{(i)}, \quad \lambda_i \geq 0, \sum\nolimits_{i=0}^{d} \lambda_i = 1 \right\}$$

□

In the following, these pairs of terms will be used interchangeably: point and 0-simplex; segment and 1-simplex; triangle and 2-simplex; tetrahedron and 3-simplex. Simplices can be combined to form complex objects; however, they must then obey constraints to form proper simplicial complexes as expressed in the following definition.

3.2.6. Definition. (*simplicial complex*) A finite set $\mathcal{C}$ of simplices forms a *simplicial complex*, if (a) for each $s \in \mathcal{C}$ the faces of s are members of $\mathcal{C}$ and (b) for each $s_1, s_2 \in \mathcal{C}$ the following condition holds: $s_1 \cap s_2 = \emptyset$ or $s_1 \cap s_2$ is a face of both s_1 and s_2. The dimension of $\mathcal{C}$ is defined as $\max_{s \in \mathcal{C}} dim(s)$. The set of points in $\mathbb{R}^3$ of the simplicial complex $\mathcal{C}$ is given by

$$points(\mathcal{C}) = \bigcup_{s \in \mathcal{C}} points(s)$$

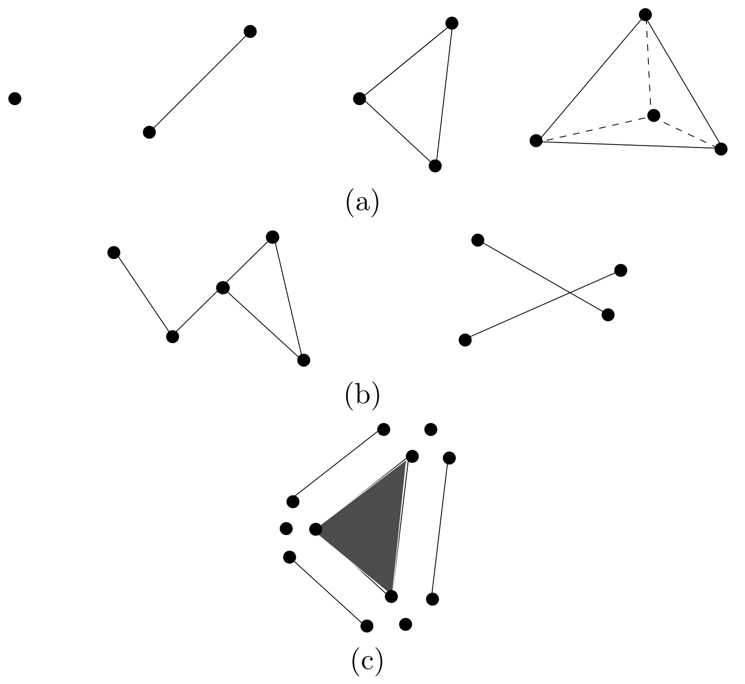


Figure 3.4: (a) simplices of dimensions 0-3, (b) left: simplices forming a simplicial complex, (b) right: simplices not forming a simplicial complex, (c) a 2-simplex and its faces.

The following restriction of the class of simplicial complexes results in spatial objects that represent simple line, areal, and volume objects.

3.2.7. DEFINITION. (*d-mesh*) Let $d > 0$. A d-dimensional simplicial complex $\mathcal{C}$ is called a *d-mesh* if (a) for every simplex $s \in \mathcal{C}$ with $dim(s) < d$ there exists an $s' \in \mathcal{C}$ such that s is a face of s', and (b) for every $(d-1)$-simplex s'' there are at most two simplices which contain s'' as faces.

A 1-mesh is also called *polyline*, a 2-mesh is also called a *triangular mesh*, and a 3-mesh is also called a *tetrahedral mesh*. Restriction (a) in definition 3.2.7 states that a polyline only contains segments, a triangular mesh only contains triangles, and a tetrahedral mesh only contains tetrahedra as simplices that are not faces of other simplices within the complex. Restriction (b) states a manifold-condition[5]: a vertex in a polyline connects at most two segments, a segment in a triangular mesh connects at most two triangles, a triangle in a tetrahedra mesh connects at most two tetrahedra. Note that tetrahedral meshes meet this requirement a priori, since the embedding space is three-dimensional.

The concepts defined so far are the basis for the spatial types, and the system GEO-TOOLKIT has been designed with these concepts in mind (Breunig, 1996; Balovnev et al., 1997). In particular, the set of spatial types comprises 0D-3D simplices (points, segments,

[5]The manifold condition implies that the vicinity of any point of the interior of a d-manifold is homeomorphic to an open subset of a corresponding point in d-space.

triangles, and tetrahedra), 1D-3D simplicial complexes (polylines, triangular meshes, and tetrahedral meshes), and “analytical objects” like lines or planes. Furthermore, one distinct type (called *Group*) allows for aggregating instances of spatial types under the simplicial complex constraints. Finally, all spatial types support geometric operations, functions, and predicates.

Although every 0D-3D simplicial complex can be represented within this model, there are the “first class citizens” polyline, triangular mesh, and tetrahedral mesh which are subject to the restrictions mentioned above. More general topologies, e.g. non-manifolds, may be represented through type *Group*. These restrictions have so far been in compliance with the requirements of several applications (Balovnev et al., 1998).

Furthermore, the following constraints on objects from this domain have to be met. First, an object corresponding to a d-dimensional simplex may not be degenerate (no flat tetrahedra are allowed, for instance, which is implied by the affine independence condition in definition 3.2.5). Second, following from the definition of a simplicial complex, any two simplices within the same simplicial complex may only intersect at common sub-simplices.

As mentioned in section 3.1.3, the design principle of separating vertices from mesh elements has been followed in the GeoToolKit design. On an implementation level, this has lead to a complex client/server storage architecture. This design principle carries over to the spatio-temporal setting.

3.2.3 Spatio-Temporal Domain

This section introduces the representational model for spatio-temporal data. It extends the spatial model of simplicial complexes and combines the characteristics of the previously mentioned models. The main properties are: (1) it allows for both continuous and discontinuous change of spatial objects; (2) the spatial domain is three dimensional; (3) it represents continuous change by *linear* vertex movement; (4) it separates vertices from mesh elements; (5) it allows for change in discretisation by time-stamping also on the simplex-level. The emerging concepts are temporal versions of the pure-spatial concepts: temporal simplices and temporal complexes, which each are interpreted as mappings from the temporal domain into the spatial domain. Furthermore, geometric constraints of the purely spatial model are injected into the spatio-temporal domain, for example, the constraint that at no instant of its lifetime a temporal triangle may degenerate to a line segment or point. Of course, a temporal simplex is allowed to degenerate at an instant outside of its lifetime, even at the boundary of one of its validity intervals if the interval is open on the respective side.

The spatio-temporal model has been designed as an extension to the pure-spatial model. Recall from section 3.1.2 that the separation of a mesh and its vertices carries over to the spatio-temporal case. Furthermore, the idea of describing change through *vertex movement* is adopted from the previous projects. Thus, the first concept to be discussed is that of vertex movement. Then, temporal simplices and temporal complexes are introduced.

A *temporal vertex* v can be defined as a function from time into three dimensional space:

$$v : \mathbb{T} \rightarrow \mathbb{R}^3$$

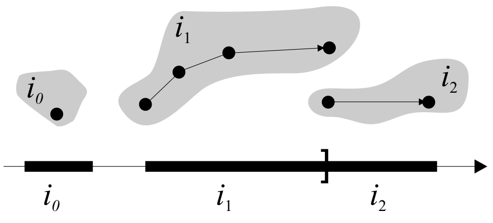


Figure 3.5: Temporal point with lifetime $e = \{i_0,\ i_1,\ i_2\}$. No movement occurs during i_0. Between i_0 and i_1 there is a temporal gap. Linear movement occurs during i_1 and i_2. The transition from i_1 to i_2 is a discontinuity. The bracket on the time-line below indicates that the location of the point for this instant is determined by the function for i_1.

where domain$(v) \in \mathbb{E}$ and v is continuous. The image of v is also called the *trajectory* of v, the curve in 3D

$$traj(v) = \{v(t) \mid t \in \text{domain}(v)\}$$

The set of such functions v is denoted by $\mathcal{T}$.

One can break up function v into two components: (1) a *position function* that describes a curve in space (the trajectory); and (2) a *velocity function* that specifies the distance travelled on the trajectory, parameterised by time. The position function must be parameterised by arclength to obtain a vertex's position at a given instant in time; then, the position is evaluated by applying first the velocity function and on the result—the arclength—the position function. However, for the scope of this thesis a simpler scheme is adopted by limiting both position function and velocity function to be piecewise linear. The implications are piecewise constant velocity and zero acceleration (except for the path vertices). More complex movement must be modelled by approximation. The subset $\mathcal{F}^{lin}$ of $\mathcal{T}$ is defined to contain exactly those functions in $\mathcal{T}$ that are piecewise linear.

For the remainder of this thesis, the representational model for vertex movement is based on $\mathcal{F}^{lin}$. Therefore, the position function is limited to be piecewise linear. This means that for each continuous trajectory T of a temporal vertex $v \in \mathcal{F}^{lin}$ there is an ordered set of points $\{p_1, \ldots, p_n\}$ such that T can be calculated through linear interpolation between two consecutive points p_i and p_{i+1}, $1 \leq i \leq n-1$. The points in the set $\{p_1, \ldots, p_n\}$ are called the *support points* of v.

Furthermore, also the velocity function, parameterised by t, is limited to be piecewise constant. This means that if vertex v is at location p_i at time t_i and at location p_{i+1} at time t_{i+1}, then the *direction* of the velocity vector of v is that of vector $p_{i+1} - p_i$. The velocity vector itself is then obtained by dividing by $(t_{i+1} - t_i)$. The location of v between these instants is obtained by

$$v(t) = p_i + \frac{t - t_i}{t_{i+1} - t_i}(p_{i+1} - p_i), \quad t \in (t_i : t_{i+1})$$

This implies constant velocity (zero acceleration) between p_i and p_{i+1}.

3.2.8. Definition. (*temporal point*) Let e denote a minimal temporal element. Furthermore, let each interval $i \in e$ be mapped to a piecewise linear, continuous function $v_i \in \mathcal{F}^{lin}$. A *temporal point* is defined as a mapping v as follows:

$$\begin{array}{rcl} v : \mathbb{T} & \to & \mathbb{R}^3 \\ t & \mapsto & \begin{cases} v_i(t) & \text{if } \exists i \in e : t \in i \\ \textit{undefined} & \text{otherwise} \end{cases} \end{array}$$

The set of these mappings v is denoted by $\mathcal{T}^{lin}$. □

To summarise, in this thesis the model of vertex movement $\mathcal{T}^{lin}$ is based on piecewise linear functions v that are given by a sequence of assignments A ordered by the time-value $time(p_i)$ assigned to each point p_i in A. These points are called the *support points* of v. Of course, points with equal coordinates may be present in S. Some support points in A mark changes in *direction*, some mark changes only in *velocity*, some mark both, and some mark discontinuities (see also figure 3.5). No other restrictions, for example physical restrictions, apply to the points in A except for those mentioned. Such restrictions have to be imposed at the application level.

For the remainder of the section, the following notation is introduced. Let $S = \{f_1, \ldots, f_n\}$ denote a finite set of functions of one variable t over a common domain. Then, $S(t)$ is a shorthand for the set

$$S(t) = \{f_i(t) | f_i \in S,\ t \in \text{domain}(f_i)\}$$

Extending purely spatial simplices and simplicial complexes, $d + 1$ temporal vertices in $\mathcal{T}^{lin}$ can be combined to form a d-simplex at every instant in their common lifetime. The following definition makes this precise (see also figure 3.6).

3.2.9. Definition. (*temporal simplex*) Let V denote a set of $d+1$ temporal vertices in $\mathcal{T}^{lin}$. Let $e \in \mathbb{E}$ be a temporal element. A *temporal d-simplex* $s[V, e]$ is the function that maps each instant in e to the convex hull of $V(t)$ as follows:

$$\begin{array}{rcl} s[V, e] : \mathbb{T} \supseteq e & \to & \mathcal{P}(\mathbb{R}^3) \\ t & \mapsto & conv(V(t)) \ = \\ & & \left\{ p \;\middle|\; p = \sum_{i=0}^{d} \lambda_i v_i(t), \quad \lambda_i \geq 0, \sum_{i=0}^{d} \lambda_i = 1 \right\} \end{array}$$

For reasons of integrity, the following conditions on $s[V, e]$ must hold: (1) the temporal vertices $v_i \in V$ must be valid during e, hence $e \subseteq time(v_i)$; (2) for all $t \in e$ the points in $V(t)$ must be affine independent. Similar to the pure-spatial case, the dimension $d = |V| - 1$ of $s[V, e]$ is denoted by $\dim s$, each proper subset of V forms a (temporal) *face* $s[V', e]$ of $s[V, e]$. □

The second condition in definition 3.2.9 states that at every instant in its lifetime, a temporal segment may not degenerate to point, a temporal triangle may not be without area, and a temporal tetrahedron may not be flat (figure 3.7). In parallel to the 3D case, a *temporal* simplicial complex can be composed of a set of temporal simplices and a temporal element e of validity.

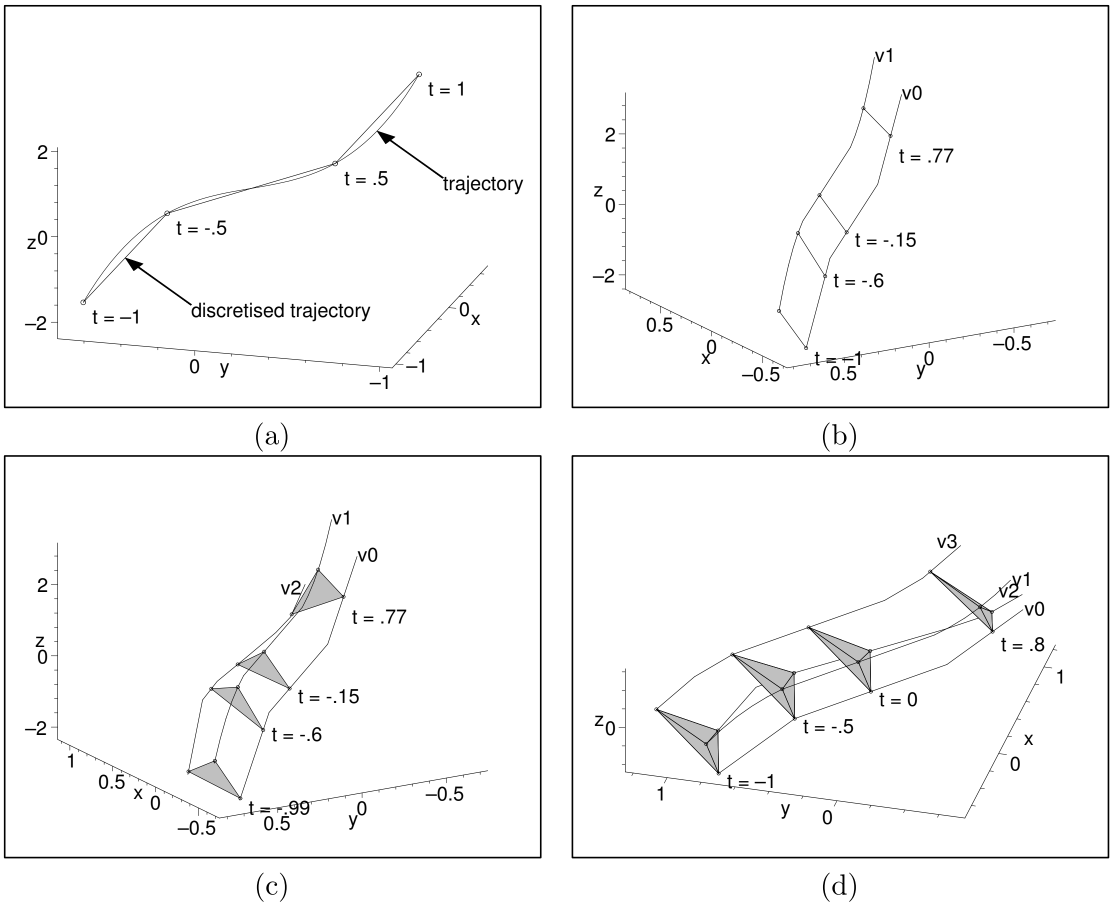
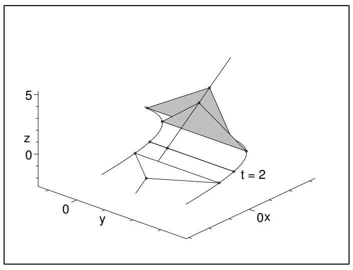


Figure 3.6: Temporal simplices. (a) A trajectory has been discretised linearly to form a temporal vertex; the vertex‘s support points are labelled by their time value. (b) Trajectory of two temporal vertices and some snapshots of the temporal segment defined by these vertices. (c) Similarly for a temporal triangle. (d) Similarly for a temporal tetrahedron.

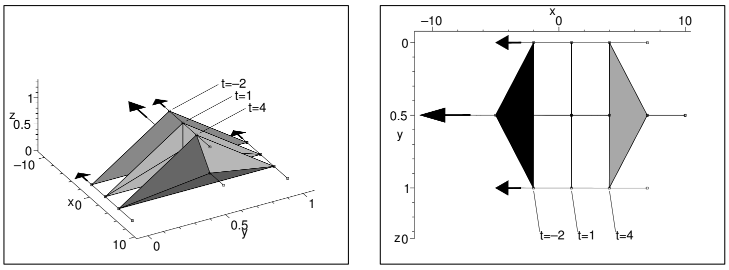
3.2.10. DEFINITION. (*temporal simplicial complex*) Let S denote a finite set of temporal simplices and $e \in \mathbb{E}$ a temporal element. A *temporal simplicial complex* $\mathcal{C}[S, e]$ is the function that maps each instant $t \in e$ to the (pure-spatial) simplicial complex $S(t)$. For reasons of integrity, the following conditions on $\mathcal{C}[S, e]$ must hold: (1) the temporal simplices $s_i \in S$ must be valid during e, hence $e \subseteq \mathit{time}(s_i)$; (2) for each instant $t \in e$ the set $S(t)$ must meet the properties of a simplicial complex (see definition 3.2.6). □

The second condition in definition 3.2.10 states that $\mathcal{C}$ is subject to the following restriction: for all $t \in I$ the complex should suffer no “self-intersection”. Parallel to the pure-spatial case the set of temporal simplicial complexes is restricted to form *temporal d-meshes* that means temporal polylines, temporal triangular meshes, and temporal tetrahedral meshes.

3.2.11. DEFINITION. (*temporal d-mesh*) Let $\mathcal{C}[S, e]$ denote a temporal simplicial complex. $\mathcal{C}[S, e]$ is called a *temporal d-mesh* if (1) for each $t \in e$ the set $S(t)$ is a valid d-mesh and



(a) temporal triangle



(b) temporal tetrahedron

Figure 3.7: Two examples for temporal simplices violating the affine-independence condition of definition 3.2.9. (a) A temporal triangle invalid at time $t = 2$. (b) A temporal tetrahedron invalid at time $t = 1$; left: side view, right: top view.

(2) for each $t, t' \in e$ the condition $\dim S(t) = \dim S(t')$ holds. □

It must be noted that—on the level of *internal* representation, focus of the next chapter—the elements of S in a temporal d-mesh $\mathcal{C}[S, e]$ are augmented by a topological information: with each temporal simplex the set of neighbouring simplices is stored. These extended simplices are also called "d-simplices with neighbourhood" and are the elements of so-called e-complexes (Breunig, 1996). The neighbourhood-information is beneficial for algorithms on meshes (Breunig, 1996; Breunig et al., 2001). In terms of storage overhead for this redundancy, there is a distinction between the pure-spatial and the temporal case. For the former, this redundancy means storing $d + 1$ references with each (pure-spatial) d-simplex; however, for the latter, the redundancy means storing a number of references with each temporal d-simplex that is not bounded by a constant. The details of storing neighbour references are subject of the next chapters.

On the way to elementary and compound operations, the definitions of *timesteps* of a spatio-temporal object are introduced, and on their basis the *input sizes* to spatio-temporal operations are defined. These concepts are used in subsequent chapters to analyse running times of algorithms for operations defined on spatio-temporal objects.

Timesteps mark instants in a spatio-temporal object's lifetime, at which something "important" is happening to it. Therefore, the timesteps of a spatio-temporal object are given as a finite subset of $\mathbb{T}$. Again, it is distinguished between a mesh and its vertices which leads to the concepts of geometry- and discretisation-timesteps.

In particular, a temporal point v changes its position through the course of time. Every interval in the lifetime of v is mapped to a piecewise linear, continuous function (definition 3.2.8), the support points of which are part of the timesteps. Let e denote the lifetime of v and v_i the function that is mapped to an interval $i \in e$.

$$\text{timesteps}(v) = \bigcup_{i \in e} \{t \mid v_i(t) \text{ is support point of } v_i\}$$

Let a temporal d-simplex, $d > 0$, be given as $s[\{p_1, \ldots, p_{d+1}\}, e]$. The set of gsteps of s is the union of timesteps of its temporal points, limited to the lifetime e of the simplex.

$$\text{gsteps}(s) = \partial e \;\cup\; \bigcup_{i=1}^{d+1} (\text{timesteps}(p_i) \cap e)$$

Secondly, through the course of time a temporal complex may not only change its location and shape of its simplices, but also its discretisation. Therefore, it is distinguished between *discretisation-* and *geometry*-timesteps, or dsteps and gsteps for short. For a temporal complex $\mathcal{C}[\{s_1, \ldots, s_n\}, e]$ the gsteps are defined as follows:

$$\text{gsteps}(\mathcal{C}) \;=\; \bigcup_{s \in \mathcal{C}} \text{gsteps}(s)$$

Furthermore, the *time-line* of a spatio-temporal object o refers to the ordered sequence of instants in gsteps(o).

Timesteps form the basis for defining input sizes to spatio-temporal operations. It is important to emphasise that the following definitions do not apply to *storage space utilisation.* Instead, they are targeted at *running times* of operations. Some of them, for example, will loop over the gsteps of a temporal simplex. Hence, the contribution to the *running time* is gsize (see below).

The input size of a temporal vertex v is measured as follows.

$$\text{size}(v) = 4 \cdot |\text{timesteps}(v)|$$

The size of a temporal d-simplex s, $d > 0$, is given by the number of point-references plus the number of intervals in the temporal element of validity.

$$\text{size}(s) \;=\; (d+1) + |e|$$

Given the separation of discretisation and geometry, one can distinguish between two different input sizes of a temporal complex, depending on whether geometric information is used or not. For instance, algorithms for detecting *holes* in a (spatial) triangular mesh do not need geometric information of their simplices[6]. While the latter can be defined easily as

$$\text{gsize}(\mathcal{C}) \;=\; |e| + \sum_{s \in \mathcal{C}} \text{size}(s)$$

the former refers to the number of neighbourhood-relationships among temporal simplices and is explained in the following. Given two temporal d-simplices s_1, s_2 and an instant t, the (spatial) simplices $s_1(t)$ and $s_2(t)$ are called *neighbours*, denoted by $s_1(t) \leftrightarrow s_2(t)$, if they share a common face, that is, $\mathit{points}(s_1(t)) \cap \mathit{points}(s_2(t)) = \mathit{points}(f)$, where f is a (d-1)-face of both $s_1(t)$ and $s_2(t)$. Figure 3.8 illustrates this relationship among simplices. A temporal d-mesh $\mathcal{C}[S, e]$ can then be seen as a (time-dependent) graph. To this end, let F denote the set of all temporal $(d-1)$-faces of the temporal simplices in S, that is $F = \{f_{\{i_1,\ldots,i_d\}}(s) \mid s \in S\}$. Furthermore, there is a special node $\circledcirc$ corresponding to the exterior of the mesh. The arcs in the time-dependent graph are of the form $[(n_1, n_2), e]$, which means that there is an arc from n_1 to n_2 for every t contained in the temporal element e. Then, the graph is given by

$$G = (N, A), \text{ where } N = S \cup F \cup \{\circledcirc\} \text{ and}$$

[6]Note, that in complexes internal topology is represented explicitly.

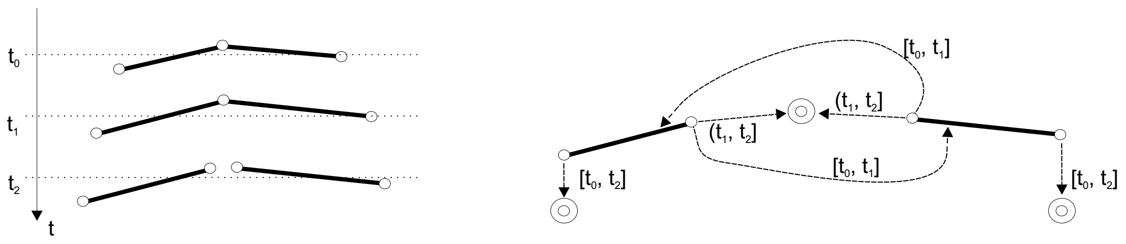


Figure 3.8: Left: A temporal polyline at three different instants. The connectivity changes after instant t_1. Right: The neighbourhood-relationships of the simplices. The special symbol ⊚ denotes the exterior of the polyline.

$$\begin{aligned} A \;:=\; & \left\{[(f_{\{i_1,\ldots,i_d\}}(s), s'), e] \mid s \neq s' \in S,\ e = \{t \mid s(t) \leftrightarrow s'(t)\} \neq \emptyset\right\} \cup \\ & \left\{[(f_{\{i_1,\ldots,i_d\}}(s), \circledcirc), e] \mid s \in S, e = \{t \mid \ \nexists s' \in S : s \neq s' \wedge s(t) \leftrightarrow s'(t)\} \neq \emptyset\right\} \end{aligned}$$

The graph structure is bipartite, since the set of arcs A only contains arcs that are directed from nodes in F to nodes in $S \cup \{\circledcirc\}$. With this graph structure, the dsteps and the dsize of a temporal mesh can be defined. Note that for a temporal element e the cardinality $|e|$ is defined as the number of intervals in e.

$$\text{dsteps}(\mathcal{C}) = \bigcup\nolimits_{[(f,s),e]\in A} \partial\, e \tag{3.1}$$

$$\text{dsize}(\mathcal{C}) = \sum\nolimits_{[(f,s),e]\in A} |e| \tag{3.2}$$

$$\text{size}(\mathcal{C}) = \text{dsize}(\mathcal{C}) + \text{gsize}(\mathcal{C}) \tag{3.3}$$

The following lemma states a lower and an upper bound for the dsize in terms of S and gsteps.

3.2.12. Lemma. *Given a temporal mesh $\mathcal{C} = \mathcal{C}[S, e]$, the following bounds on dsize$(\mathcal{C})$ hold:*

$$\begin{aligned} \textit{dsize}(\mathcal{C}) &= \Omega\left(\sum\nolimits_{s\in S} |\textit{time}(s)|\right) \\ \textit{dsize}(\mathcal{C}) &= O\left(|\textit{gsteps}(\mathcal{C})| \cdot |S|^2\right) \end{aligned}$$

Proof: The lower bound can be seen as follows. Every face $f \in F$ of a simplex $s \in S$ participates in at least one arc $[(f, s), e]$. The definition of A implies that for every interval $i \in$ time(s) there is either a separate arc in A or the temporal element of an arc contains i. Either case contributes to the dsize of the mesh. More arcs or bigger temporal elements are possible, but not less. Since every d-simplex in a temporal d-mesh contributes $d+1$ faces and d is bounded by a constant, the lower bound holds. Furthermore, the bound is tight, that means, meshes exist which have this dsize. This is possible, for instance, if there are only arcs from f to ⊚. In such a case, every simplex in S is isolated from every other simplex in the mesh. Every face of a simplex s contributes exactly one arc to A with the temporal element time(s).

For the remainder of the discussion it is assumed that an arc

$$[(f, n), e = \{i_1, \ldots, i_n\}] \in A$$

is replaced by the set of arcs

$$[(f, n), i_1], \ldots, [(f, n), i_n]$$

Obviously, the replacement yields the same result, since then

$$\text{gsteps}(\mathcal{C}) = \sum_{[(f,s),e] \in A} |e| = |A|$$

The set gsteps($\mathcal{C}$) partitions the temporal evolution of a mesh such that between two consecutive instants t_1, t_2 from this set every vertex of the mesh performs a linear move during $[t_1, t_2]$. Furthermore, every face $f \in F$ can "connect" to $|F| - (d + 1)$ different faces during this interval, yielding as much arcs in A. However, f can also connect to the special node $\circledcirc$. Hence, the worst case is if there are alternate connections of f to a face and to $\circledcirc$, contributing $2 \cdot (|F| - (d + 1))$ arcs to A. More arcs for f would be possible only, if f could connect to another face again during $[t_1, t_2]$. But this is prevented by the linearity of the vertices's moves: once a connection is ended, it cannot occur again during $[t_1, t_2]$. Therefore, in total there are at most

$$(|\text{gsteps}(\mathcal{C})| - 1) \cdot |F| \cdot 2 \cdot (|F| - (d + 1))$$

arcs in A. Since $|F| = (d + 1)|S|$, the upper bound holds. □

3.3 Discussion

This section discusses and explores important properties of the model defined in the previous section. The first aspect is to what extent the model is capable of building spatio-temporal objects. It is discussed why specific constraints are built into the model while others are not. Strongly related is the second problem of constructing spatio-temporal objects out of existing ones by applying set-based operations union, intersection, and difference. Therefore, the model is examined for closure under these operations.

3.3.1 Separation of Representation and Construction

The model is representational only. Neither does it contain physical constraints, nor are there any restrictions for the transition from one timestep to the next that go beyond the injected spatial restrictions. As a consequence, it is left unspecified how to construct a spatio-temporal object, e.g., out of different (purely spatial) snapshots, or how to refine approximation. Application-dependent as such construction operations are, it is arguable whether they should be tightly integrated into a database model, and the assumption for this work is that they are to be maintained on an application's level (compare Koubarakis et al., 2003, sect. 4.1).

In the following four examples, it is described how the discretisation of a surface may change as time proceeds and thereby modelling continuous change. In the first example, a triangle s can be replaced by a set M of two or more triangles at a timestep. The new triangles cover the area of s exactly: $points(M) = points(s)$. In the second example, the surface can be extended by adding a triangle s' at the boundary (cf. figure 3.9). This can be accomplished by assigning s' the left-open temporal interval $(t_1, t_x]$. Thus, s' does not exist at t_1, but at

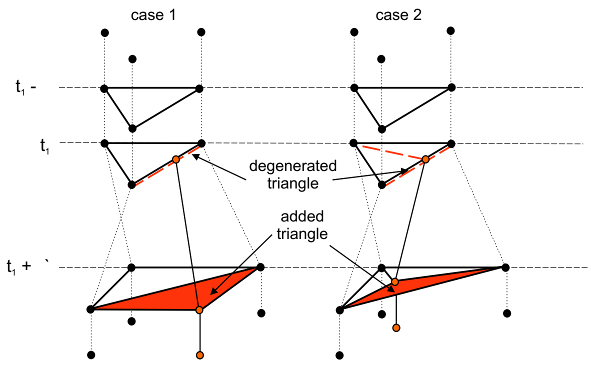


Figure 3.9: An insertion of a new time-dependent simplex.

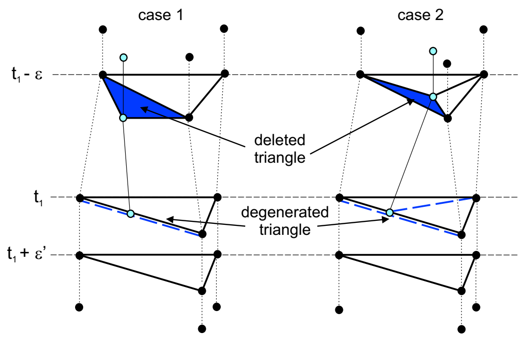


Figure 3.10: A deletion of a time-dependent simplex.

$t_1 + \epsilon$. If extrapolated to t_1, s' would become a degenerated triangle that corresponds to a boundary segment of the surface. Hence, the geometry of the surface is consistent at t_1 and extends "smoothly" as time proceeds to t_2. In the third example a triangle s'' vanishes (cf. figure 3.10). This case is inverse to the previous one. The temporal interval of s'' is right-open and if continued to t_1 would also reduce to a line segment. The last example deals with changes to (external) topology. A "hole" may evolve when a moving vertex v is placed on a non-boundary line-segment at a timestep t_i which then moves into the area of triangle s. The temporal interval of s is $[t_x, t_i]$. Let the vertices of s be v_0, v_1, v_2, then s is replaced by two new triangles (v_0, v_1, v) and (v_1, v_2, v) each of which has the temporal interval $(t_i, t_y]$.

It must be noted that the examples above are purely descriptive and that a mathematical formulation of how to achieve these surface changes is not part of the model. However, as was stated above, a methodology for geometry evolution is beyond the tasks of a database model and also not adequate: a methodology for one application domain may not fit a further application domain. Therefore, the model is purely representational, in that it is *capable* of representing continuous change types as given in the examples above.

Furthermore, it is easy to verify that the proposed model also conforms to the taxonomy of basic spatio-temporal processes according to Claramunt et al. (1997). Every basic spatio-temporal process (compare table 1.1, p. 3) can be represented: (1) stability, since the trajectory of a temporal vertex can degenerate to a point yielding no movement at all; (2) deformation, since the trajectories of temporal vertices can be specified independently; (3) expansion/contraction, since the trajectories of temporal vertices can be the result of a *scaling function*; (4) rotation for a *rotation function*, linearly approximated; (5) translation for a *translation function* applied to every temporal vertex in a temporal mesh. Additionally, also change in topology can be represented, although more demanding on the application‘s level that must formulate a mathematical model to compute such changes.

3.3.2 Non-Closure of Set-Based Operations

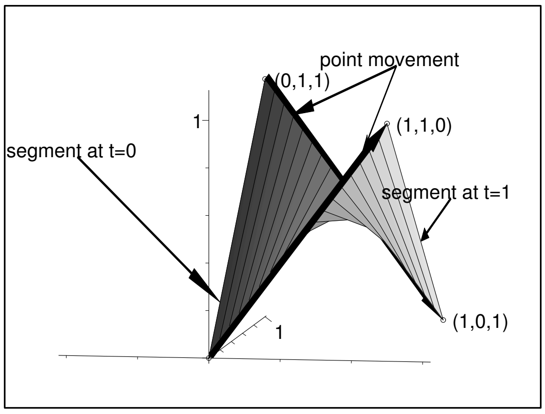
Regarding the representable spatio-temporal objects as time-dependent *point sets*, it is worth examining the closure under set-based operations union ($\cup$), intersection ($\cap$), and difference ($\setminus$). Conceptually, the definition of these operations is extended straightforwardly for the spatio-temporal setting. Given two spatio-temporal objects o_1 and o_2, the result of such an operation is a function of time, the domain of which is the intersection of the two input domains of o_1 and o_2. For example, the union operation can be defined as follows:

$$o_1 \cup_{\mathbb{T}} o_2 : \ t \mapsto o_1(t) \cup o_2(t), \ \text{where domain}(o_1 \cup_{\mathbb{T}} o_2) = \text{time}(o_1) \cap \text{time}(o_2)$$

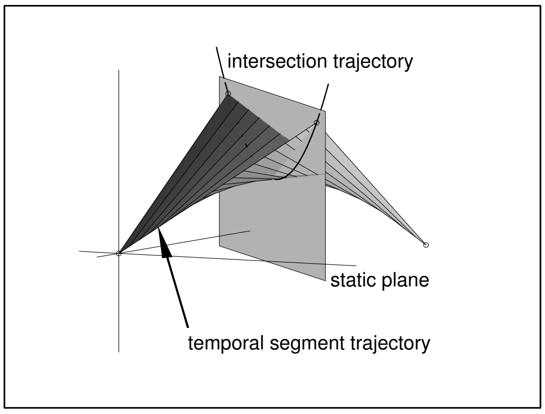
Intersection and difference can be defined in the same manner. While any spatio-temporal object—as defined in the previous section—is a time-dependent point set, not every point set corresponds to a spatio-temporal object. The notion of a time-dependent point set is more general. Hence, the set of time-dependent point sets is partitioned into two classes, where the first class contains those point sets that can be represented by a spatio-temporal object and the second class contains those point sets that cannot be represented by a spatio-temporal object as defined in the previous section. For this reason, it is interesting to see if operations on spatio-temporal objects can “lead out” of the first class, that is, if operations are *closed* under certain operations. More concrete, since these point sets have the representation given above, *closure* means here that the result of a set-based operation on any two given point sets *under the given representation* can always be brought into this representation. If there are cases where the result of such an operation cannot be represented within the model, then the model is not closed under this particular operation. The operation *difference* is—in a strict sense—not closed, because the representational model cannot express point sets without boundary (*open* point sets). Within the field of computer-aided design, a related observation has lead to so-called *regularised* operations that eliminate also isolated points or dangling edges. Hence, the set-based operations are interpreted in their regularised form:

$$o_1 \setminus^*_{\mathbb{T}} o_2 : \ t \mapsto o_1(t) \setminus^* o_2(t) = \text{closure}(\text{interior}(o_1(t) \setminus o_2(t)))$$

The other operations are defined analogously. While the problem of open sets, isolated points, etc., can be overcome in this way, the situation is different with spatio-temporal *intersection* and linear vertex movement $\mathcal{T}^{lin}$ as defined in section 3.2.3. By proving that an intersection can yield non-linear vertex-movement, the negative result of non-closure is reported.



(a)



(b)

Figure 3.11: To the proof of proposition 3.3.1. (a) Trajectory of a temporal segment. (b) Intersection with a static plane resulting in a temporal vertex with non-linear trajectory.

3.3.1. PROPOSITION. *The set of temporal simplicial complexes with the $\mathcal{T}^{lin}$ model of vertex movement is not closed under intersection.*

Proof: It must be shown that there are temporal simplices the intersection of which cannot be expressed in terms of the model. To this end, it suffices to construct an example based on a temporal segment. The temporal vertices v_1, v_2 of this segment are defined as follows:

$$\begin{aligned} v_1 : \mathbb{T} \supset [0,1] &\rightarrow \mathbb{R}^3 \\ 0 &\mapsto (0,0,0) \\ 1 &\mapsto (1,1,0) \\ v_2 : \mathbb{T} \supset [0,1] &\rightarrow \mathbb{R}^3 \\ 0 &\mapsto (0,1,1) \\ 1 &\mapsto (1,0,1) \end{aligned}$$

Hence, the first vertex performs a linear move from the origin to $(1,1,0)$, the second vertex performs a linear move from $(0,1,1)$ to $(1,0,1)$ (see figure 3.11(a)). The trajectory of the so defined temporal segment is a *bilinear interpolation* of the aforementioned points: a curved surface. Therefore, intersecting the temporal segment with a static triangle can result in moving vertex trajectory that is non-linear (figure 3.11(b)). Indeed, only if in this case the triangle is placed such that it lies within a plane parallel to either $xy-$ or $yz-$plane, a linear vertex movement results; however, all other placements would result in a non-linear movement. □

Hence, to create a temporal complex by intersection, the resulting vertex trajectories must be approximated piecewise linear. A similar result for the two-dimensional space has been reported by Chomicki and Revesz (1999a). However, the result presented here is stronger

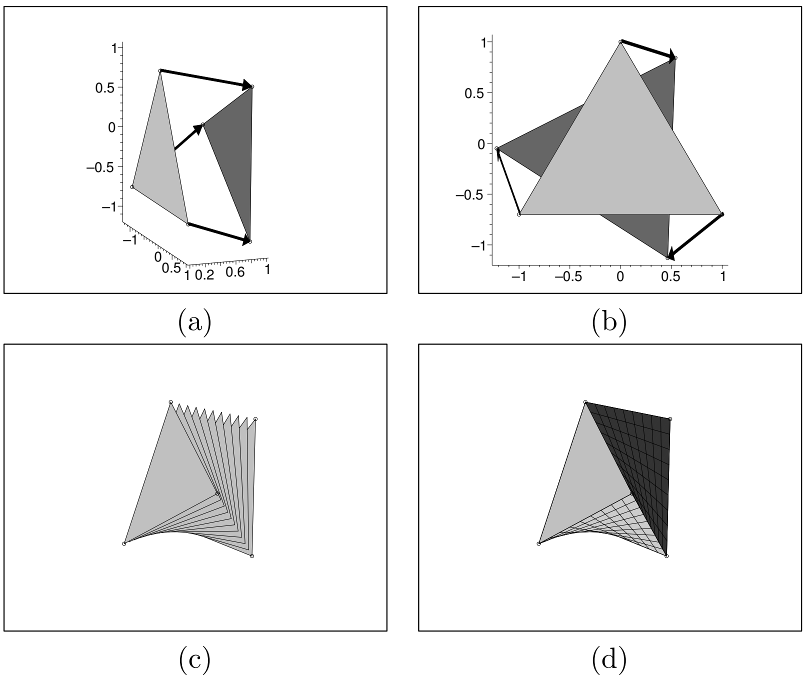


Figure 3.12: Example for a temporal triangle whose trajectory forms a solid with non-linear bounding faces.

in the following sense. It shows that for three dimensions the intersection operation is not closed, even for the case of intersecting a spatio-temporal object with a *pure-spatial* object. An implication of the proof above is that other operations of interest are also affected, for example the trajectory-operator. Given a spatio-temporal object o, this operation results in a pure-spatial object corresponding to o‘s trajectory. While in two dimensions, the model would support this operation, the situation is different in three dimensions, as exemplified in figure 3.12. Here, the trajectory of a temporal triangle forms a solid with non-linear bounding faces that cannot be represented within the model.

Chapter 4

Design of Data Types for a Spatio-Temporal Database System

The representational model forms the basis for data types, the integration of which into a database system is the centre of interest in the following two chapters. These types are to be used in application development or by ad-hoc query systems. One can distinguish between the *external* level that defines the appearance of the type system to the user, and the *internal* level that forms its implementation. While the latter is discussed in the next chapter, the former is in the focus now.

The design of the spatio-temporal data types is guided by several objectives. Among such objectives is *simplicity* and *adequacy*, leading to a type system that subsumes pure-spatial and spatio-temporal objects under a common data type and keeps thus the number of types and interfaces small. A further important design objective stems from the *part/subpart*-relationship between a mesh (the part) and its simplices (the subparts). Here, the spatio-temporal data types are designed such that they allow for the *sharing* of subparts. Additionally, this relationship is strongly connected to a *logical storage hierarchy* for spatio-temporal objects.

The spatio-temporal data types are embedded in the object-oriented framework. Since objects are characterised by state *and* behaviour, both aspects are discussed. In particular, concerning *behaviour* it is also investigated how to transfer pure-spatial predicates and operations to the spatio-temporal setting. However, the discussion is limited to operations for which implementation concepts are developed in this thesis. Concerning the *state* of an object, not all (potential) states are valid for spatio-temporal objects. Therefore, so-called *restrictions* are defined on the new types so that spatio-temporal objects comply with the definitions from the previous chapter.

The type system extends the set of predefined classes of ObjectStore$^{\text{TM}}$, the underlying persistent object storage management system, by spatio-temporal classes to be offered in user-level schema modelling. Thereby, object-oriented features shall be made available, for instance, substitution polymorphism; a further design objective to be followed.

The remainder of the chapter is organised as follows. The first section reviews concepts of the object-oriented framework that are being used subsequently. Then, the design objectives for the spatio-temporal type system are derived. The logical storage hierarchy is focus of the following section. After that, the classes for spatio-temporal data types are presented, ranging from the pure-temporal classes to classes for temporal meshes. Part of the description are structure, behaviour, and object restrictions.

4.1 Object-Oriented Framework

The design of the spatio-temporal data types follows the object-oriented paradigm. This choice is due to the fact that the GeoToolKit projects (compare section 3.1) were mainly conducted within the environment of ObjectStore$^{\text{TM}}$, an object storage system. Hence, the system GeoToolKit serves as a starting point, although the concepts derived here are independent from this environment to a high degree. As current object-relational database systems also offer many object-oriented features, several aspects might carry over to such systems.

The remainder of the chapter makes use of the following concepts and notations. Since the system is built *on top* of ObjectStore$^{\text{TM}}$, its data modelling capabilities are readily available. Central is the notion of a *class*, instances of which are called *objects*. Thereby, every class comprises two parts: (1) a part describing the *state* of its instances; (2) a part describing to what messages instances react (given by *method signatures*) and how they do it (given by *method implementations*).

Every class corresponds to a type, while the reverse does not hold. Each type is defined by the set of (potential) values for that type and the set of operations that can be applied to them. Certain *built-in* types are available, like *integer*, *bool* or *float*, along with their usual operations and sets of values. The structure of a class c with the attributes $a_1, \ldots, a_n$ of types $t_1, \ldots, t_n$, respectively, is denoted by $c = [a_1 : t_1, \ldots, a_n : t_n]$. The set of values for c, denoted by $\mathcal{V}(c)$, is defined by

$$\mathcal{V}(c) = \{(\ldots, v(a_{i_1}), \ldots, v(a_{i_n}), \ldots) \mid v(a_{i_j}) \in \mathcal{V}(t_{i_j})\}$$

which is the set of all tuples having attribute values for the attributes $a_1, \ldots, a_n$ from $\mathcal{V}(t_i)$, respectively. This construct replaces the Cartesian product of values for the purpose of inheritance (Watt, 1990). The fact that class c is a subclass of class c' is denoted by $c <: c'$ (following Abadi and Cardelli, 1996). Then, with the principle "inheritance is subtyping" one obtains the inclusion relationship: $c' <: c \Rightarrow \mathcal{V}(c') \subseteq \mathcal{V}(c)$. As usual, for a given object o of class c, the value for the attribute a_i of o is denoted by the so-called *dot-notation* $o.a_i$. Instances of classes are objects o that carry an implicit unique identifier, denoted by o.id.

Furthermore, it is assumed that *type constructors* are available for sets (Set<T>), lists (List<T>), and references (Ref<T>), where T is called the *parameter* of the type constructor. Ref<T> models a reference to an object of type T. An object r of type Ref<T> can assume the value null. The dereference operator is denoted by resolve: Ref<T> $\rightarrow T$. Dereferencing a null-reference raises an exception. In C++-parlance, the shorthand $*o$ is used here for resolve(o), the shorthand o->a is used for resolve(o).a for an object o having an attribute a. Methods are treated analogously.

The notion of equality of objects is used at several places. Here, it denotes structural equality, if not otherwise stated. Hence, given two objects o_1, o_2 of type T with the set of attributes A, equality of o_1 and o_2, denoted by $o_1 = o_2$, is recursively defined as $o_1.a = o_2.a$ for each a in $A \setminus \{id\}$. This is in contrast to the identity of two objects o_1, o_2, denoted by $o_1.id = o_2.id$. Equality of two given references r_1, r_2 holds if and only if they refer to the same object, that means $r_1 = r_2 :\Leftrightarrow r_1\texttt{->}id = r_2\texttt{->}id$.

Following the principle "inheritance is subtyping" (Abadi and Cardelli, 1996) and the subsumption principle, any spatio-temporal data type also has the type of its abstract base class *stObject* (defined below), such that it is possible, for example, to form database collections of spatio-temporal objects that are of diverse (most specific) types. Furthermore, the mechanism of dynamic dispatch for method invocation (*late binding*) is available. Furthermore, certain methods of *stObject* have parameters that are also of type *stObject*. Given two references to *stObject*-objects o and o', such a method m is called as follows: $o\texttt{->}m(o')$. Here, it is necessary to determine the most specific types of both o and o', since this information is needed for the implementation (see definition of the methods below). While the most specific type of o can be determined at runtime via dynamic dispatch of method m, the most specific type of o' can be determined through runtime type information (rti); however, rti can be circumvented by the following alternative approach that has also been applied in the GeoToolKit-projects (Balovnev et al., 2003). The approach works as follows. Let the most specific type of o and o' be A and B, respectively. Then, the call $o\texttt{->}m(o')$ must be resolved to the method $A::m(\text{Ref}<B>)$ or, alternatively, $B::m(\text{Ref}<A>)$, since the methods under consideration are commutative (see definitions below). First, the method call is resolved via late binding to the method $A::m(\text{Ref}<\textit{stObject}>)$. The latter method is implemented such that it forwards the method call to its final destination by calling $o'\texttt{->}m((\text{Ref}<\text{A}>)o)$. Finally, again via late binding, the target method $B::m(\text{Ref}<\text{A}>)$ is reached. For this to work, the mentioned methods must be implemented in the specified way for all combinations of arguments.

Normally, every value of $\mathcal{V}(c)$ can be assumed for an object of class c. However, if some values of $\mathcal{V}(c)$ are not admissible, class c is said to be *restricted*. The restrictions can usually not be checked at compile-time; rather they must be assured at runtime. In the following, two types of restrictions are distinguished. First, there are *enforced* restrictions, which means that non-admissible values are converted into admissible values (see class *temporalElement* below for an example). Second, there are *exception* restrictions, which means that non-admissible values raise an exception to be resolved by an appropriate handler.

4.2 Design Objectives

The design objectives for the spatio-temporal data types on the user-level are as follows. First, spatial and spatio-temporal types should be integrated in a *transparent* way in that there is no distinction between the two kinds of data on this level. The reason lies in *simplicity* and *adequacy* of the type system. It yields simplicity in that the number of types and operations is small and hence less complex for the user. It yields adequacy in that nevertheless both kinds of data can be represented and treated uniformly, since spatial data is modelled as a special case of spatio-temporal data (compare the embedding of spatial data in section 3.2.2). As a consequence, it is refrained from general type constructors, since the meaning of a construct (interpolation function, etc.) like "*temporal*$<T>$" for an arbitrary type T can be given on

Design Objective	Justification
Direct support of temporal d-meshes, more general topologies through a special data type	Emphasises most important kind of spatial data to be extended
Transparent integration: subsume spatial types as special cases of spatio-temporal types	Yields simplicity and adequacy
Build dynamic data structures	Supports geometric modelling
Separate mesh from vertices	Saves space, avoids redundancy, enables building several meshes from the same set of vertices
Allow for object-oriented features	Use modelling capabilities of the underlying framework

Table 4.1: Overview of the mentioned design objectives for spatio-temporal data types (details see text).

an application‘s level only and is hidden to the DBMS. To sum up, spatio-temporal types are not modelled as instantiable template classes, but as (simple) classes and thereby subsume pure-spatial types as special cases of spatio-temporal types.

A second design objective is to support the temporal d-meshes (see section 3.2.3) through specific types and operations, while more general topologies should be represented by a container type conforming to the geometric constraints of a temporal simplicial complex mentioned in section 3.2.3. This is in tradition with the GeoToolKit approach for spatial types (Balovnev et al., 1997).

Building *dynamic* data structures is a further objective the design should follow. This is justified by the requirement to support geometric modelling tasks that need to update spatio-temporal objects after their initial creation (see also section 3.1.1 and the fourth specification “time-stamp update” in chapter 1). Therefore, the spatio-temporal types should contain operations for insertion, removal, and update of “subparts“ like vertices, mesh elements, etc. (see below).

A further design objective stems from the separation of a mesh and its vertices. For the reasons given in section 5.6, the task of *storage allocation* for vertices and mesh elements is transferred to the application‘s level, to some degree. Therefore, creation- and insert-operations should have a means to specify the storage area for an object‘s subparts. While this design objective violates *simplicity*, an implementation can supply default storage areas and thus relieve the user of this task.

Furthermore, the objective to support the geometric constraints (see section 3.2.3) should be followed. This leads to the specification of restrictions defined on the new classes. For example, a pure-spatial type for triangular meshes, say TM, consists—among others—of a set of triangle objects. Following $\mathcal{V}(TM)$, any set of triangle objects can be used to form a TM object. However, the definition of a triangular mesh (definition 3.2.7) restricts the admissible sets of triangle objects, for instance in such a way that any two triangles in the set may only intersect at sub-simplices. This condition on sets of triangle objects must be checked at runtime, the class TM is therefore restricted.

Finally, users should be able to take advantage of object-oriented features with the new spatio-temporal data types. This objective is not "automatically" available, since further goals on the internal level conflict with this objective, for example, substitution polymorphism. Such a feature is needed in situations, where one wants to insert triangle objects into a triangular mesh whose types are in fact a *subtype* of type triangle, for instance, to extend triangle objects with a user-specified attribute. At the same time, it is desirable to separate the geometry-part from the attribute-part for clustering that would support queries based on geometry. This separation, in conjunction with an assembly where needed, does not come for free; however, the design described below facilitates the use of object-oriented features, while their realisation is discussed in the next chapter.

A summary of these design objectives is shown in table 4.1. One implication of the objectives is that spatio-temporal objects are organised hierarchically and are also interdependent. This hierarchy is focus of the following section.

4.3 Logical Storage Hierarchy

To allow for the flexibility inherent in the design objectives of the previous section, spatio-temporal objects are organised in a logical storage hierarchy. The hierarchy is termed *logical*, since it corresponds to the user‘s view upon the storage structure of spatio-temporal objects and is in contrast to its internal realisation (see next chapter). The properties of the hierarchy are threefold. First, it reflects the *part/subpart*-relationship (aggregation) of spatio-temporal objects. For instance, a d-mesh is comprised of d-simplices. Second, it allows for a *sharing* of temporal simplices and temporal vertices, thereby modelling n:m-relationships among these objects. Third, it enables the allocation hints and thereby the grouping of temporal points and simplices in the persistent storage area.

The structure of the logical storage hierarchy in terms of the *part/subpart*-relationship is depicted in figure 4.1 and can be described as follows. The top of this hierarchy forms the container type *space* (not shown) that holds references to spatio-temporal objects. On the next level there are the elements of a *space*, the temporal d-meshes that are logically containers of references to temporal d-simplex objects. On the next level there are temporal d-simplices that hold references to temporal point objects. Finally, the lowest level consists of the referred point objects storing "the geometry" of the spatio-temporal objects.

However, the aggregation-relationship cannot be modelled as an *existential* aggregation, which would imply a 1:n-relationship between meshes and vertices, hindering the sharing of vertices. As a consequence, it must be possible to establish an n:m-relationship between meshes and vertices. Similarly, meshes must be capable of sharing their simplices. This modelling approach of sharing subparts results in the use of references as opposed to embedded objects, as described in the next section.

Finally, using certain storage classes, a user can build groups of vertices and groups of simplices that are clustered internally for faster retrieval and allow for data partitioning, for instance onto separate storage devices. Details of internal organisation are subject of the next

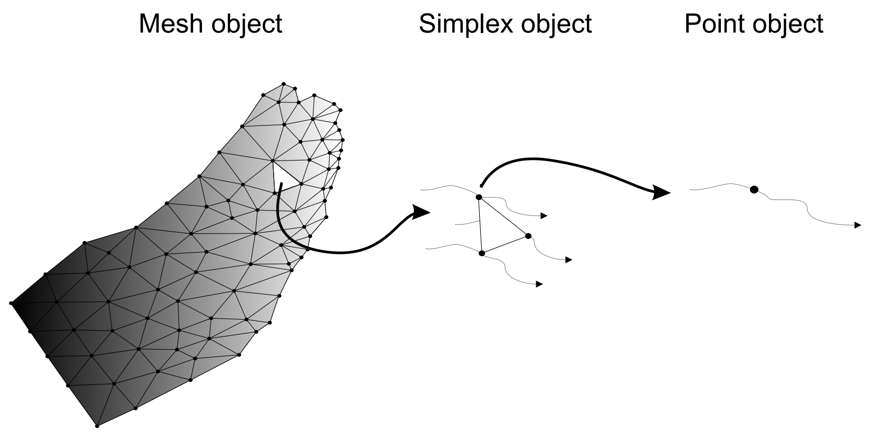


Figure 4.1: Logical view on the storage hierarchy of spatio-temporal data types. The bold arrows indicate reference-relationships.

chapter.

To sum up, the proposed hierarchy allows for flexibility. Temporal points are separated from temporal simplices, temporal simplices are separated from temporal meshes. Points and simplices can thus be shared among several of the respective container objects.

4.4 Storage Classes

The simplex objects can reside in different places in persistent memory and they are managed by special data structures. The point manager is responsible for storing and retrieving point objects. The segment-, triangle-, and tetrahedron-managers are responsible for the respective objects. Though described in detail in the next chapter, here, instances of the data structures are used as parameters that specify the placement of simplex objects. It is worth noting that an application can create an arbitrary number of such managers to facilitate data partitioning.

4.5 Data Types for Spatio-Temporal Objects

The classes to be presented are organised in an inheritance tree, the root of which forms a so-called *abstract base class*. Then, for each $d = 0, \ldots, 3$, there is a pair of classes corresponding to the notion of a d-simplex and the notion of a d-mesh, respectively, that are subclasses of the abstract base class. Furthermore, the class *group* corresponds to the more general notion of a temporal simplicial complex. This design yields the objective of *direct* support of d-simplices and d-meshes, while also allowing for more general temporal simplicial complexes.

To present the system less complex to the user, the number of classes is kept small. This is facilitated by the fact that pure-spatial data types are modelled as a special case of spatio-temporal data types. As a result, for each $d = 0, \ldots, 3$, there is only one class corresponding to the notion of a d-simplex, capturing both temporal and pure-spatial objects. Likewise, for each $d = 1, \ldots, 3$, there is only one class corresponding to the notion of a d-mesh. Importantly, this design does not rule out the usage of pure-spatial *internal representations* along with, for example, proven pure-spatial geometric algorithms or indexes. The question of how to transfer pure-spatial operations to the spatio-temporal setting is discussed in the next section. To sum up, the design of treating pure-spatial objects as a special case of spatio-temporal objects yields the objective of a transparent integration of pure-spatial and spatio-temporal objects.

In the following, an overview of the user-level classes is given. First, the pure-temporal classes are introduced, followed by the spatio-temporal types and the user-level storage classes. See also figure 4.7 for a UML-representation of the class hierarchy.

4.5.1 Pure-Temporal Types

The pure-temporal classes are used to represent the (temporal) validity of objects. In particular, regarding spatio-temporal objects as functions from the time domain into the spatial domain, temporal elements are used to specify the domain of these functions. First, a class representing instants is described. Instants are used to construct intervals to be described thereafter. Finally, the class for temporal elements is presented.

Class *instant*

Objects of this class model (temporal) instants. Hence, $\mathcal{V}(\textit{instant})$ corresponds to the temporal domain as defined in section 3.2.1. It contains the elements of a finite representation of the set of real numbers and the special symbols $-\infty$, ∞, and **undef**. The class offers usual arithmetic operators with the following signature.

$$+, -, \cdot, \div : \textit{instant} \times \textit{instant} \rightarrow \textit{instant}$$

It is assumed that the operations have the "usual" semantics of the underlying representation scheme for real numbers. Special care must be taken for operands that assume the values of the special symbols. The comparison operator

$$<: \textit{instant} \times \textit{instant} \rightarrow \textit{bool}$$

is specified to raise an exception if one of the operands equals **undef** (see also definition 3.2.1).

Class *interval*

Objects of class *interval* model temporal intervals. Recall from definition 3.2.2 that an interval consists of a lower and an upper boundary and that it can be open on one or both sides and may degenerate to a single instant. Hence, the class appears to the user in the following way:

$$\textit{interval}{=}[\text{start: } \textit{instant}, \text{ end: } \textit{instant}, \text{ leftopen: } \textit{bool}, \text{ rightopen: } \textit{bool}]$$

Since $\mathcal{V}(interval)$ shall correspond to the set $\mathbb{I}$, the class *interval* is subject to exception restrictions. These restrictions follow those mentioned in definition 3.2.2. Hence, for any object i of type *interval* the following conditions hold:

1. Either both i.start and i.end equal **undef** or none equals **undef**, and
2. if i.start $\neq$ **undef**, then i.start $\leq$ i.end, and
3. If i.start $=$ i.end then i.leftopen=i.rightopen=**false**

These conditions are to be checked during construction or updates of objects of class *interval*. The constructor has the following signature.

$$\text{new: } instant \times instant \times bool \times bool \rightarrow interval$$

The parameters are used to assign the start-, end-, leftopen-, and rightopen-attribute, respectively. Further operations are available for extracting the boundary of *interval* objects. The first operation applied on an object i of class *interval* returns i.start, the second returns i.end. The signature is as follows.

$$\text{inf, sup: } interval \rightarrow instant$$

Special treatment for the interval **never** leads to returning **undef**. The *closure* operation applied on an object i of class *interval*

$$\text{closure: } interval \rightarrow interval$$

constructs an interval i' that is the closure of i (definition 3.2.2). In particular, the following conditions hold: i.start $=i'$.start, i.end $=$ i'.end, and i'.leftopen $=$ i'.rightopen $=$ **false**. There is no special treatment for the interval **never**.

Furthermore, Allen‘s thirteen qualitative relationships between intervals, extended for open and closed intervals, are available as predicates. The signature is given by:

$$interval \times interval \rightarrow bool$$

The semantics of the predicates can be taken from figure 4.2. If one of the operands has the value **never**, the operands are incomparable with respect to these predicates. The treatment is therefore to raise an exception.

The set $\mathbb{I}$ is not closed under set-based operations, as discussed in section 3.2.1, and this holds also for $\mathcal{V}(interval)$. Although the intersection-operation forms an exception, it is refrained from offering set-theoretic operations for class *interval*, thereby following the simplicity principle. The mentioned intersection-operation can be applied by conversion to an object of class *temporalElement* and applying its corresponding methods.

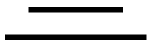
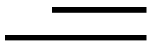
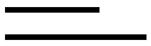
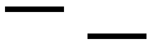
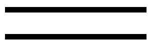
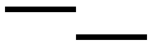
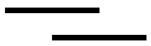
Predicate	Schematic view	Semantic extended by points
i during j		$(i^- > j^- \vee (i^- = j^- \wedge i^{(} \wedge j^{[})) \wedge$ $(i^+ < j^+ \vee (i^+ = j^+ \wedge i^{)} \wedge j^{]}))$
i finishes j		$(i^- > j^- \vee (i^- = j^- \wedge i^{(} \wedge j^{[})) \wedge$ $(i^+ = j^+ \wedge (i^{]} \Leftrightarrow j^{]}))$
i starts j		$(i^- = j^- \wedge (i^{[} \Leftrightarrow j^{[})) \wedge$ $(i^+ < j^+ \vee (i^+ = j^+ \wedge i^{)} \wedge j^{]}))$
i before j		$i^+ < j^- \vee$ $(i^+ = j^- \wedge \neg(i^{]} \wedge j^{[}))$
i equal j		$(i^- = j^- \wedge (i^{[} \Leftrightarrow j^{[})) \wedge$ $(i^+ = j^+ \wedge (i^{]} \Leftrightarrow j^{]}))$
i meets j		$i^+ = j^- \wedge i^{]} \wedge j^{[}$
i overlaps j		$(i^- < j^- \vee (i^- = j^- \wedge i^{[} \wedge j^{(})) \wedge$ $i^+ > j^- \wedge$ $(i^+ < j^+ \vee (i^+ = j^+ \wedge i^{)} \wedge j^{]}))$

Figure 4.2: Allen‘s thirteen relationships among intervals, extended for intervals with points (i.e. open, semi-open and degenerate intervals are allowed). Only seven relationships are shown, since the remaining ones are inverses of the others (except for *equal*). The upper interval is denoted by i, the lower interval by j. The lower boundary of an interval i is denoted by i^-, the upper boundary by i^+. The fact that i is open (closed) on the lower boundary is denoted by $i^{(}$ ($i^{[}$). The case for the upper boundary is handled analogously.

Class *temporalElement*

Objects of this class model temporal elements, which are defined as (arbitrary) sets of intervals. Hence, the attribute part of the class appears to the user in the following way:

$$\textit{temporalElement} = [\text{intervals: Set}<\textit{interval}>]$$

An object e in $\mathcal{V}(\textit{temporalElement})$ is subject to the following enforced restriction (compare section 3.2.1):

$$\{\ i \,|\, i \in \text{e.intervals}\ \} = \min\ \{\ i \,|\, i \in \text{e.intervals}\ \}.$$

This means that any temporal element is represented in its minimal form; a fact that is of interest to the user, since intervals are changed or even removed to obtain the minimal form (*coalescing*). This also entails the design decision that the class‘s Set<*interval*> does not hold references to intervals. Rather, intervals are *copied* into the temporal element such that objects external to a temporal element remain unchanged during coalescing. Note that this approach breaks with substitution polymorphism as the user can insert objects of most specific type *interval* only.

Three constructors are available. First, there is the default constructor that creates a *temporalElement* object e with e.intervals=$\emptyset$. Secondly, the copy constructor takes as argument an object e' of type Ref<*temporalElement*> and creates an object e such that e.intervals=e'.intervals. Thirdly, there is the constructor that takes as argument an interval i and constructs a *temporalElement* e such that e.intervals = $\{\{i\}\}$.

new:		$\rightarrow$ *temporalElement*
new:	Ref<*temporalElement*>	$\rightarrow$ *temporalElement*
new:	*interval*	$\rightarrow$ *temporalElement*

The set $\mathbb{E}$ is closed under set-based operations, and this holds also for $\mathcal{V}(\textit{temporalElement})$. As a result, the following operations are available, accompanied by operations for constructing the complement, hull, and closure of temporal elements. Definitions of these operations have been given in chapter 3.

$\cap, \cup, -$:	*temporalElement* $\times$ *temporalElement*	$\rightarrow$ *temporalElement*
complement:	*temporalElement*	$\rightarrow$ *temporalElement*
hull:	*temporalElement*	$\rightarrow$ *temporalElement*
closure:	*temporalElement*	$\rightarrow$ *temporalElement*

4.5.2 Extension of Pure-Spatial Predicates

Treating pure-spatial objects as special cases of spatio-temporal objects implies a unified definition of *operations* on such objects. As a consequence, one must extend the definition of pure-spatial operations to the spatio-temporal setting adequately. One approach to achieve this has been reported by Güting et al. (2000). This approach is called *lifting* and has been described in chapter 2.

However, since type constructors for expressing the continuous change of values are not assumed in this thesis, the *lifting*-approach cannot be followed directly. The reason for this lies in the *return type* of operations, which do not have a correspondence here, for example a time-dependent boolean value. Therefore, a signature of a binary spatial predicate, which is given by

$$\textit{spatial} \times \textit{spatial} \longrightarrow \textit{bool}$$

in the pure-spatial setting, carries over to the spatio-temporal setting, but the operation must be defined anew carefully, while retaining its meaning when applied to pure-spatial objects.

To extend pure-spatial predicates to the spatio-temporal setting, two different alternatives appear meaningful. See also Erwig and Schneider (2002) for a related discussion on this topic. To be more concrete, the remainder of the discussion will be based on the binary predicate *intersects* which is defined to be true for two given spatial objects o_1, o_2 if and only if o_1 and o_2 have at least one point $p \in \mathbb{R}^3$ in common. The first alternative for extension lies in a *for-all-* or *always*-semantic, the second one in an *exists*-semantic. It is argued that the *always*-semantic is not appropriate for the extension of a pure-spatial predicate, while the extension under an *exists*-semantic can be well-defined.

- *Always*-semantic. Here, it is assumed that the (spatio-temporally-extended) *intersects* predicate evaluates to *true* if and only if for all $t \in \mathbb{T} \setminus \{\textbf{undef}, -\infty, \infty\}$: $o_1(t)$ spatially-intersects $o_2(t)$. Of course, this approach is not well-defined, since the lifetime of spatio-temporal objects is usually not equal to $\mathbb{T} \setminus \{\textbf{undef}, -\infty, \infty\}$. One had to extend the pure-spatial *intersects* predicate to undefined spatial objects also. A better way is to "bind" the predicate to the common lifetime of o_1 and o_2. Then the meaning of the predicate would be described as *"Did the objects intersect during their whole common lifetime?"* Hence, the spatio-temporally-extended *intersects* predicate under the *always*-semantic is defined as follows.

$$o_1 \ \textit{always-intersects} \ o_2 \ :\Leftrightarrow$$

$$\{t \in \mathrm{dom}(o_1) \cap \mathrm{dom}(o_2) \mid o_1(t) \ \textit{intersects} \ o_2(t)\} = \mathrm{dom}(o_1) \cap \mathrm{dom}(o_2)$$

  However, as an undesirable consequence, this expression yields *true* for objects with disjoint lifetimes. One way out could be to set it for this case to one of the values *true*, *false*, or *undefined*. Setting to *true* or *false* is obviously of little sense. Therefore, the only reasonable extension under the *always*-semantic is as follows. Let D denote $\mathrm{dom}(o_1) \cap \mathrm{dom}(o_2)$.

$$o_1 \ \textit{always-intersects} \ o_2 \ :\Leftrightarrow$$

$$\Leftrightarrow \begin{cases} \{t \in D \mid o_1(t) \ \textit{intersects} \ o_2(t)\} = D & \text{if } D \neq \emptyset \\ \textit{undefined} & \text{otherwise} \end{cases}$$

  This solution, however, is not fully satisfactory, since a three-valued logic is necessary for a realisation (the predicate may evaluate to *true*, *false*, and *undefined*).

- *Exists*-semantic. The extension of a pure-spatial predicate under the *exists*-semantic is oriented towards the question *"Was there an instant at which both objects existed and also intersected?"* More formal, one can define the extension as follows. Let D again denote $\mathrm{dom}(o_1) \cap \mathrm{dom}(o_2)$.

  $$o_1 \textit{ sometime-intersects } o_2 \;:\Leftrightarrow \{t \mid t \in D \text{ and } o_1(t) \textit{ intersects } o_2(t)\} \neq \emptyset$$

  The defined predicate evaluates to *false* if D is empty. Any extended pure-spatial predicate will evaluate to *false* if applied on two objects that have disjoint lifetimes. This is adequate, since under the *exists*-semantic one asks for the existence of an instant at which the (pure-spatial) predicate holds. If no such instant exists, the extended predicate evaluates to *false.*

To sum up, in the following it is assumed that pure-spatial predicates are extended under the *exists*-semantic. It is worth noting that the extended predicates retain their pure-spatial meaning if applied to (embedded) pure-spatial objects.

4.5.3 Abstract Base Class *stObject*

The abstract base class *stObject* is the root of the inheritance tree containing the classes for spatio-temporal objects. Three aspects must be investigated. First, there is the integration into the inheritance tree of the type system. Second, there are the attributes that comprise class *stObject.* Third, there is the set of methods, or the *interface* of class *stObject.* The interface comprises only those methods for which implementation concepts are developed in this thesis. Further operations are possible, see for instance (Galton, 1995; Muller, 1998; Erwig and Schneider, 1999; Hazarika and Cohn, 2001; Erwig and Schneider, 2002)

The integration into the inheritance tree of the type system is performed as follows. Class *stObject* is chosen as an abstract base class for the classes that model spatio-temporal objects. Class *stObject* itself does not inherit from any class. The inherent aspects of temporality and spatiality of a spatio-temporal object are included through *aggregation*, as described below.

Put differently, the (object-oriented) modelling question is whether an *stObject consists of* a temporal element (aggregation), or whether it *is a* temporal element (inheritance). However, the latter must be ruled out for the following reason. The approach implies that methods for *temporalElement* are also part of the interface of class *stObject.* But the meaning of operations for *temporalElement*-objects is different from those for spatial or spatio-temporal objects. For example, invoking the method "intersects" on a given *stObject*-object could perform the intersection test of the temporal elements only, while not testing the spatio-temporal extent, or it could perform the intersection test for the spatio-temporal extent, while not testing the temporal element, or, finally, it could perform the intersection test for both. As a result, the alternative approach yields a conflict of methods. For these reasons, the conception that an *stObject has* a temporal element of validity is justified and this approach is adequate.

The following attributes comprise class *stObject.* Common to all spatio-temporal objects—whatever its concrete kind—are: (1) the attribute *temporalElement* that specifies the lifetime of the object; and (2) the spatial extent of the object for every instant in its lifetime. Whereas

Method name	**Parameter types**	**Return type**
Retrieval Methods:		
lifetime:		→ *temporalElement*
at:	*instant*	→ Ref<*stObject*>
range_query:	*boundBox4D*	→ Ref<*stObject*>
range_query:	*plane*	→ Ref<*stObject*>
range_query:	*half-space*	→ Ref<*stObject*>
Boolean Predicates:		
intersects:	Ref<*stObject*>	→ *bool*
disjoint:	Ref<*stObject*>	→ *bool*
Metric Function:		
min_dist:	Ref<*stObject*>	→ *scalar*
Temporal Functions:		
when_intersects:	Ref<*stObject*>	→ *temporalElement*
when_disjoint:	Ref<*stObject*>	→ *temporalElement*
when_min_dist:	Ref<*stObject*>	→ *temporalElement*
Update Methods:		
lifetime:	*temporalElement*	→

Figure 4.3: Summary of the method signatures for the abstract base class *stObject*. Definitions see text. The table shows only those methods for which implementation concepts are developed in this thesis.

the former can be modelled in the abstract base class, the latter remains specific to the concrete classes that inherit from *stObject*. Hence, the attribute-part of the class appears to the user in the following way:

$$\mathit{stObject} = [\text{lifetime: } \mathit{temporalElement}]$$

It is worth noting that the attribute is itself not stored with *stObject* (though the methods manipulating it are). The reason is that for atemporal objects this attribute is not stored at all, since it is implicitly set to $(-\infty, \infty)$.

The interface of class *stObject* can be described as follows (see also figure 4.3). Methods common to spatio-temporal objects are specified in this base class, while each method is redefined in the subclasses and accessed via dynamic dispatch (late binding) during runtime. As a result, each method described below is a virtual method to be defined in the more specific classes. The methods for *stObject* are categorised as follows: (1) methods for retrieval that result in an object of type *stObject*; (2) methods for Boolean predicates whose return type is *bool*; (3) methods for metric functions, here the minimum Euclidean distance resulting in a *scalar*; (4) methods for temporal functions that result in a *temporalElement*; (5) methods for updates. Those methods that construct objects, allocate them in transient memory.

The first category (retrieval methods) comprises a snapshot query and a spatio-temporal range query. The snapshot query "at" returns an object valid at a single instant. Class *stObject* can specify the "temporal part" of this method, while the spatial part is specific for each subclass. Applied on an *stObject* o and an *instant* i, the temporal part of the method is defined as follows. If $i \notin o$.lifetime, it raises an exception indicating that i is outside the lifetime of o. Otherwise, it constructs an *stObject* o' with o'.lifetime = new *temporalElement*(*interval*(i, i, **false**, **false**)).

The next method of the first category (retrieval methods) is the spatio-temporal range query that takes as argument a *boundBox4D* object, a *plane* object, or a *half-space* object. Applied on an *stObject* o and a parameter Q, the method retrieves the part of o that intersects Q.

Second, there are spatio-temporal predicates the return type of which is *bool*. For the scope of this thesis, non-trivial Boolean predicates are limited to a "contains"-, an "intersects"-, and a "min_dist_within"-relationship between spatio-temporal objects, as defined below. While more diverse topological relationships between spatial objects could be extended to the spatio-temporal setting under the exists-semantic and with the means developed in chapter 6, the field of qualitative spatio-temporal reasoning is an ongoing field of research (see also Galton (1995); Muller (1998); Erwig and Schneider (1999); Hazarika and Cohn (2001); Erwig and Schneider (2002)). The implementation aspect remains still a challenge.

As noted above, the Boolean predicates are interpreted in their exists-semantic (section 4.5.2). The discussion is limited to the intersects- and disjoint-predicates. The definitions of these predicates are based on well-known predicates for atemporal spatial objects. The predicate "intersects" for two atemporal spatial objects o and o' holds if and only if there is a point $p \in \mathbb{R}^3$ contained in o that is also contained in o'. Given two *stObject*-objects o_1 and o_2, the available methods are defined as follows. Let D denote o_1.lifetime $\cap$ o_2.lifetime and let i denote an *interval*-object, then

$$o_1.\text{intersects}(o_2) \quad :\Leftrightarrow \quad \begin{cases} \textbf{true} & \text{if } \{t \mid t \in D \text{ and } o_1(t) \textit{ intersects } o_2(t)\} \neq \emptyset \\ \textbf{false} & \text{otherwise.} \end{cases}$$

$$o_1.\text{disjoint}(o_2) \quad :\Leftrightarrow \quad \begin{cases} \textbf{true} & \text{if } \neg\, o_1.\text{intersects}(o_2) \\ \textbf{false} & \text{otherwise.} \end{cases}$$

The description of the computation of these predicates is deferred until chapter 6.

The third category of *stObject*-methods comprise metric functions, here limited to the Euclidean distance between two spatio-temporal objects. Applied on two *stObject*-objects o_1 and o_2, method o_1.distance(o_2) computes the *scalar*-value

$$\min \{\text{distance}(o_1(t), o_2(t)) \mid t \in o_1.\text{lifetime} \cap o_2.\text{lifetime}\}$$

If the intersection of o_1's lifetime with o_2's lifetime is empty, method "distance" raises an exception. Again, the description of the computation of this function is deferred until chapter 6.

The previous methods of the third and fourth category (Boolean predicates and metric function) are also available in their *when*-form, and they make up the fifth category of *stObject*-methods. The *when*-form of these methods result in a *temporalElement* that shows at what instants the predicates yield true (for the predicates above) or at what instants the distance has been minimal (for the Euclidean distance function).

Finally, the category of update methods for class *stObject* comprises only method "lifetime" that updates the valid time (attribute "lifetime") of an *stObject*-object. Since subclasses must ensure restrictions on this attribute, the method must be redefined in the subclasses and must therefore be virtual.

To sum up, class *stObject* is an abstract base class that itself does not inherit from any class and that consists of a *temporalElement*-attribute. It contains the interface common to the specific classes that model spatio-temporal objects and that inherit from class *stObject*. Every method is a virtual method that can be applied on any object of type *stObject*. The signatures of the methods that have been defined thus far are summarised in figure 4.3. The implementations of the methods are part of the subclasses of *stObject* and a method $o.m(o', \ldots)$ is accessed at runtime (1) via dynamic dispatch for the object o the method is called upon and (2) via the mechanism described in section 4.1 for the parameter o'. Methods that construct objects, allocate such objects in transient memory. Further details for particular subclasses of *stObject* are focus of the following sections.

4.5.4 Spatio-Temporal Simplices

Each notion of a temporal 0-, 1-, 2-, and 3-simplex is represented by a separate class. Hence, there is one class for temporal points, one class for temporal segments, one class for temporal triangles, and one class for temporal tetrahedra. All these classes inherit from their abstract base class *stObject*. This section describes these classes covering the aspects of attributes, restrictions, and methods.

Before discussing classes for spatio-temporal simplices, further support classes must be introduced. First, there is the class *scalar* that models a scalar value from $\mathbb{R}$. Second, there are the classes *point3D* and *point4D* that model an element of the underlying space $\mathbb{R}^3$ and a time-stamped point in $\mathbb{R}^3$, respectively:

$$\begin{aligned} \textit{point3D} &= [\text{x: } \textit{scalar}\text{, y: } \textit{scalar}\text{, z: } \textit{scalar}] \\ \textit{point4D} &= [\text{x: } \textit{scalar}\text{, y: } \textit{scalar}\text{, z: } \textit{scalar}\text{, t: } \textit{instant}] \end{aligned}$$

Third, there is the class *pl_curve* (piecewise linear curve). This class models a piecewise linear function from $\mathcal{F}^{lin}$ that maps each instant in $\mathbb{T} \setminus \{\textbf{undef}, -\infty, \infty\}$ to an element in $\mathbb{R}^3$ (see section 3.2.3). Hence, a *pl_curve*-object consists of a set of *point4D* objects that are the support points of the curve. Such a support point [x, y, z, t] corresponds to the location of the point at an instant: $t \mapsto (x, y, z)$.

$$\textit{pl\_curve} = [\text{traj: Set}<[\text{s: } \textit{point4D}]>]$$

The restriction on each object v of type *pl_curve* is such that

1. $|v.\text{traj}| \geq 2$. This restriction is needed, since at least two support points are needed to define a linear function.

2. For all s in v.traj: $s.\text{t} \notin \{\textbf{undef}, -\infty, \infty\}$.

Method name	**Parameter types**	**Return type**
Constructors/Destructors:		
new:		$\rightarrow$ Ref<*point*>
new:	Set<[*interval*, *pl_curve*]>	$\rightarrow$ Ref<*point*>
new:	Ref<*point*> $\times$ Ref<*pointManager*>	$\rightarrow$ Ref<*point*>
destroy:	option	$\rightarrow$
Update Methods:		
set:	*instant* $\times$ *point3D*	$\rightarrow$

Figure 4.4: Methods for *point*, showing only methods that differ from those for *stObject*.

3. For $s_1, s_2 \in v.\text{traj} : \; s_1.\text{t} = s_2.\text{t} \Rightarrow s_1 = s_2$. This restriction ensures that v.traj is a *function* from time into space.

Finally, there is the class *boundBox* that models an axis-aligned, three-dimensional bounding box: *boundBox* = [lowerLeft: *point3D*, upperRight: *point3D*]. Its main purpose is to provide arguments for range queries.

Class *point* <: *stObject*

The class *point* corresponds to a temporal 0-simplex and is used to model a time-dependent vertex that assumes values from $\mathbb{R}^3$.

$$point = [\text{lifetime: } temporalElement, \text{ m: lifetime} \rightarrow pl\_curve]$$

The attribute "lifetime" is inherited from class *stObject* and denotes the valid time of a *point*-object. A new notation is introduced by "m: lifetime → *pl_curve*" that denotes a mapping from each element in "lifetime" to a *pl_curve* object (compare definition 3.2.8).

In addition to the methods inherited from class *stObject*, the following methods are available for class *point*. They comprise the constructors and the destructor to create and delete objects of class *point*, as well as the update method "set" to change the trajectory of a *point*-object (see also figure 4.4).

The first constructor is the so-called default constructor (i.e. has no arguments). It constructs an "empty" *point* object that corresponds to null for every instant. The second constructor listed in figure 4.4 receives as parameter a set of interval/pl_curve-pairs. This set represents the mapping from a temporal interval to a piecewise linear curve. Implicitly, the lifetime of the temporal point object is specified by the intervals in the set. These intervals must form a minimal temporal element, otherwise the constructor raises an exception. The third constructor listed in figure 4.4 denotes the copy constructor that takes as parameter a reference to a *point*-object and to a *pointManager*-object. The reference to the former may not be null, otherwise the constructor raises an exception. The constructor copies the data of its parameter into the second parameter. If that is null, the constructor uses the same *pointManager* as the “this”-object. Method “destroy” is the destructor for a point object. There are two options for the parameter “option”: (1) DEFAULT: the point is deleted, if and only if there are no simplex objects holding references to it; (2) TRIGGER_CASCADE: the point object is deleted, every simplex-object holding references to it is deleted and removed from every referring mesh.

Method “set” is used to alter the trajectory of the *point*-object. Hence, it takes as parameters an instant and a point in $\mathbb{R}^3$.

Classes for temporal 1-, 2-, and 3-simplices

The temporal d-simplices ($d > 0$) hold references to $d{+}1$ temporal points and carry a temporal element of validity that is inherited from class *stObject*. Furthermore, *stObject*‘s methods are subject to late binding (virtual methods). Therefore, the implementation of each method is redefined in the subclasses; for brevity, most methods are not repeated here, while their particular implementation is focus of a separate chapter. To start with, the class *segment* <: *stObject* corresponds to a temporal 1-simplex. It consists of a temporal element of validity and two references to temporal points. Its structure is defined as follows. Attribute “v” is meant here as a shorthand for “vertex”.

$$\textit{segment} = [\text{lifetime: } \textit{temporalElement}, \text{ v: Ref}<\textit{point}>[2]]$$

Class *segment* is subject to the following restrictions. Let s denote an object of type *segment*. First, each $s.\mathrm{v}[i]$ may not be null. Second, s must be temporally sound, that means s.lifetime must be contained in each $s.\mathrm{v}[i]$->lifetime. Third, the affine independence constraint on a temporal segment s leads to the following restriction. For all instants in s‘s lifetime, s‘s vertices may not coincide, that means, for all $i \in s$.lifetime: $s.v[0]\text{->}at(i) \neq s.v[1]\text{->}at(i)$.

The class *triangle* <: *stObject* corresponds to a temporal 2-simplex. Its structure is defined as follows.

$$\textit{triangle} = [\text{lifetime: } \textit{temporalElement}, \text{ v: Ref}<\textit{point}>[3]]$$

Class *triangle* is subject to the following restrictions. Let t denote an object of type *triangle*. First, each $t.\mathrm{v}[i]$ may not be null. Second, t must also be temporally sound, that means t.lifetime must be contained in each $t.\mathrm{v}[i]$->lifetime. Third, the affine independence constraint on a temporal triangle t leads to the following restriction. For all instants in t‘s lifetime, t‘s three vertices may not lie on a common line.

The class *tetrahedron* <: *stObject* corresponds to a temporal 3-simplex. Its structure is defined as follows.

$$\textit{tetrahedron} = [\text{lifetime: } \textit{temporalElement}, \text{ v: Ref}<\textit{point}>[4]]$$

Class *tetrahedron* is subject to the following restrictions. Let t denote an object of type *tetrahedron*. First, each $t.\mathrm{v}[i]$ may not be null. Second, t must also be temporally sound, that means t.lifetime must be contained in each $t.\mathrm{v}[i]$->lifetime. Third, the affine independence

Method name	Parameter types	Return type
Constructors/Destructors:		
new:		$\to$ Ref<*simplex*>
new:	*temporalElement* × Ref<*point*>[2/3/4]	$\to$ Ref<*simplex*>
new:	Ref<*simplex*> × Ref<*simplexManager*> × Ref<*pointManager*>	$\to$ Ref<*simplex*>
destroy:	option	$\to$
Update Methods:		
set:	Ref<*point*>[2/3/4]	$\to$

Figure 4.5: Methods for *segment/triangle/tetrahedron* that augment the methods for *stObject*. For shortness, the methods are listed in a single, combined table only; hence, "*simplex*" must be replaced by one of *segment/triangle/tetrahedron*. Likewise, "2/3/4" must be replaced by "2" in case of *segment*, by "3" in case of *triangle*, by "4" in case of *tetrahedron*.

constraint on a temporal tetrahedron t leads to the following restriction. For all instants in t's lifetime, t's four vertices may not lie on a common plane.

To sum up, the restrictions on temporal simplices comprise: (1) the restriction that none of the references to *point*-objects may be null; (2) the restriction that a temporal simplex be temporally sound; and (3) the restriction that the affine independence constraint of definition 3.2.9 must be met. Both the first and the second property are easily computed; however, the solution to the problem of checking the latter property is deferred until chapter 6.

To complete the description of the classes *segment, triangle*, and *tetrahedron*, the set of their methods is specified. First, virtual methods of the abstract base class *stObject* are inherited in each of the classes *segment, triangle*, and *tetrahedron*. Being general, the definitions given for class *stObject* also apply to those mentioned here. Therefore, the solution to the problem of implementing these methods for every class will be deferred until the next chapters. Second, there are methods for the classes *segment, triangle*, and *tetrahedron* that augment the interface of *stObject*. These methods comprise constructors and a destructor, as well as the method "set" to update an object of the particular class.

The methods are summarised in figure 4.5. The first constructor is the default constructor that creates an "empty" simplex object. The second constructor listed in figure 4.5 takes as parameters the temporal element of validity for the corresponding object and an array of references to point objects. The constructor checks the restrictions for the particular class (see above) and raises an exception in the negative case. The third constructor listed in figure 4.5 is the copy constructor that takes as parameter a reference to an object of the particular class and references to a *simplexManager* and to a *pointManager*. The first parameter may not be null, otherwise the constructor raises an exception. The two latter arguments specify where to allocate the newly constructed simplex and the copied point objects. If both are passed as null, a new simplex object is created in the same *simplexManager* as the "this"-object, which refers to its (existing) point objects. Operation "destroy" is the destructor for a simplex object. For the parameter "option" there are two options: (1) DEFAULT: the simplex is deleted, if and only if there are no references from mesh-objects; (2) TRIGGER_REMOVE:

the simplex is removed from every referring mesh object and deleted afterwards.

Method “set” takes as argument an array of references to *point*-objects that is assigned to the data member “v”. Like the first constructor, it checks the restrictions and raises an exception in the negative case.

Having described classes for temporal points, segments, triangles, and tetrahedra, the next section discusses classes for spatio-temporal meshes.

4.5.5 Spatio-Temporal Meshes

Next discussed are classes for temporal $1-, 2-, 3-$meshes. These classes also inherit from class *stObject* and carry therefore their own temporal element of validity. Furthermore, they appear to the user in such way that they contain a set of temporal simplices. Furthermore, the member-relationship between this set and its temporal simplices is also time-stamped, as will be shown below. The remainder of the section looks at classes for spatio-temporal meshes in more detail.

To start with, class *polyline* <: *stObject* corresponds to a temporal 1-mesh. It consists of the inherited temporal element of validity and a set of time-stamped references to *segment*-objects. The following class structure consists of: (1) the “lifetime” attribute for the validity of the object; and (2) the “segments” attribute that is a set of *segment*/*temporalElement*-pairs to express which segment is part of the *polyline* and when it is part of the *polyline*.

polyline = [lifetime: *temporalElement*,
segments: Set<[seg: Ref<*segment*>, timestamp: *temporalElement*]>]

Class *triangleNet* <: *stObject* corresponds to a temporal 2-mesh. Again, only references to *triangle*-objects are stored, while sub-simplices are implicit. The following class structure consists of: (1) the “lifetime” attribute for the validity of the object; and (2) the “triangles” attribute that is a set of *triangle*/*temporalElement*-pairs to express which triangle is part of the *triangleNet* and when it is part of the *triangleNet*.

triangleNet = [lifetime: *temporalElement*,
triangles: Set<[tris: Ref<*triangle*>, timestamp: *temporalElement*]>]

Class *tetraNet* <: *stObject* corresponds to a temporal 3-mesh. As in the previous two classes, sub-simplices are represented implicitly. The following class structure consists of: (1) the “lifetime” attribute for the validity of the object; and (2) the “tetrahedra” attribute that is a set of *tetrahedron*/*temporalElement*-pairs to express which tetrahedron is part of the *tetraNet* and when it is part of the *tetraNet*.

tetraNet = [lifetime: *temporalElement*,
tetrahedra: Set<[tets: Ref<*tetrahedron*>, timestamp: *temporalElement*]>]

Class *group* <: *stObject* corresponds to a (general) temporal simplicial complex. This class can also be used to store sub-simplices explicitly. The following class structure consists of:

(1) the "lifetime" attribute for the validity of the object; and (2) the "elements" attribute that is a set of *stObject*/*temporalElement*-pairs to express which stObject is part of the *group* and when it is part of the *group*.

group = [lifetime: *temporalElement*,
elements: Set<[objs: Ref<*stObject*>, timestamp: *temporalElement*]>]

The restrictions of the above classes can be described as follows. First, there is the restriction that each reference in the *segments*/*triangles*/*tetrahedra*/*elements* data member may not be null. Second, there is the restriction that every object be temporally sound. For example, for a *polyline*-object p this means that for all $[s, e]$ in p.segments the following condition holds: s->lifetime contains e. The remaining classes are analogously. Third, there is the restriction that for any instant in an object's lifetime, the d-mesh property (simplicial complex property for *group*) must hold: the segments of a *polyline* may intersect only at the segments's end points, the triangles of a *triangleNet* may intersect only at the triangles's bounding segments, the tetrahedra of a *tetraNet* may intersect only at the tetrahedra's bounding triangles, the simplices of a *group* may intersect only at common sub-simplices.

To complete the description of the classes *polyline, triangleNet, tetraNet*, and *group*, the set of their methods is specified. First, virtual methods of the abstract base class *stObject* are inherited in each of the classes *polyline, triangleNet, tetraNet*, and *group*. As for the temporal simplex classes, the definitions given for class *stObject* also apply to those mentioned here. Therefore, the solution to the problem of implementing these methods for every class will also be deferred until the next chapters. Second, there are methods for the classes *polyline, triangleNet, tetraNet*, and *group* that augment the interface of *stObject*. These methods comprise constructors, as well as the methods "insert" and "remove" to update an object of the particular class. The methods are summarised in figure 4.6. The first constructor listed in figure 4.6 takes as parameter the temporal element of validity for the corresponding object to initialise data member "lifetime". The set data member is initialised to the empty set. The second constructor listed in figure 4.5 is the copy constructor that takes as parameter a reference to an object of the particular class. This reference may not be null, otherwise the constructor raises an exception. The third constructor is the copy constructor that takes as argument a reference to mesh object, which not be null. The two parameters Ref<*simplexManager*> and Ref<*pointManager*> specify where to allocate simplex objects and point objects, respectively. If the former is null, the copy constructor inserts references to the existing simplex objects of the "this"-mesh (and thereby also the existing point objects). if this parameter is non-null, the copy constructor copies the referenced simplices into the given *simplexManager* and—if the parameter Ref<*pointManager*> is non-null— copies the point objects into the given *pointManager*. Newly created simplex objects refer to the newly created point objects.

Methods "contains" and "when_contains" check for containment of the simplex in the mesh. Method "insert" takes as argument an object of the element type of the mesh. Otherwise, it checks the restrictions for the particular class (see above), and it raises an exception in the negative case. In the positive case, the argument is inserted into the mesh. If the object is already contained in the mesh, its temporal element e of containment is adapted according to the argument e': $e \Leftarrow e \cup e'$. Finally, method "remove" takes as parameter a reference r to

Method name	**Parameter types**	**Return type**
Constructors/Destructors:		
new:	*temporalElement*	→ Ref<*mesh*>
new:	Ref<*mesh*>	→ Ref<*mesh*>
new:	Ref<*mesh*> × *level*	→ Ref<*mesh*>
destroy:		→
Membership Methods:		
contains:	*simplex*	→ *bool*
when_contains:	*simplex*	→ *temporalElement*
Update Methods:		
insert:	*simplex* × *temporalElement*	→
remove:	*simplex*	→
Index Creation/Removal:		
create_index:	*split_fct*	→
remove_index:		→

Figure 4.6: Methods for *polyline/triangleNet/tetraNet* that augment the method set for *stObject*. For shortness, the methods are listed in a single, combined table only; hence, "*mesh*" must be replaced by one of *polyline/triangleNet/tetraNet*. The "*simplex*" must be replaced by *segment* for *polyline*, by *triangle* for *triangleNet*, by *tetrahedron* for *tetraNet*, and by *stObject* for *group*.

the element type of the particular set data member. It removes all those elements $[o, e]$ from the set data member with $r = o$.[1]

Method "create_index" is used to create a spatio-temporal index (section 5.5) for the simplices contained in the mesh. The index is used to speed up operations on the mesh and the checks for update validity. The parameter *split_fct* to the method is a function that computes a split[2] when given a simplex object. Method "remove_index" is used to remove an index from the mesh.

In this chapter, interface aspects for the spatio-temporal classes have been discussed. The following chapters concentrate upon implementation aspects. Many of the problems faced have only non-trivial solutions.

[1]Remember from section 4.1 that equality of two given references holds if and only if both refer to the same object.

[2]Defined in section 5.5.2.

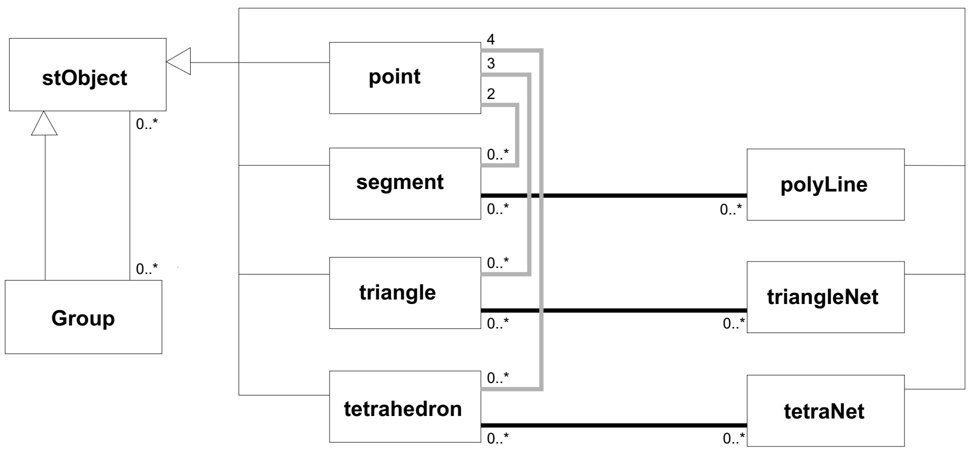


Figure 4.7: UML-representation of the hierarchy for spatio-temporal data types. The diagram reflects the classes‘s user-perspective, while “hiding” the internal representation, as explained in section 5.

Chapter 5

Design of Data Types on the Internal Level

Having described the data types on an application development‘s level, the design of an *internal* representation is focus of this chapter. Since the storage structure is built upon the object storage system ObjectStore$^{\text{TM}}$, the chapter opens with a description of the model of persistent storage and briefly describes those ObjectStore$^{\text{TM}}$ features that are used subsequently. Then, objectives for a design are derived that go beyond scalability and efficiency. Strongly related to all aspects is indexing. Not only does it support updates and range queries, but also clustering spatio-temporal objects on disk. Therefore, the chapter continues with an investigation of indexing spatio-temporal objects that includes also an overview of related work. After that, an architecture is proposed that meets the design objectives. In particular, methods for *storage*, *updates*, and *basic retrieval* are presented. An important aspect, especially when faced with updates, is that the data structures support connected components of temporal meshes. To this end, the chapter generalises the notion of a graph and transfers these problems onto the new graph structure. In particular, temporal variants of the breadth-first-search and the union-find-problem are derived. Methods for storage comprise an architecture of persistent structures for spatio-temporal objects. Algorithms for update operations modify spatio-temporal objects. The chapter closes with an algorithm for basic retrieval (the snapshot query). The fact that the elements of a temporal mesh are traversed by a breadth-first-traversal and that the point objects are clustered, exploit locality of reference by pre-fetching point-data into a point cache.

5.1 Architectural Model of Persistent Storage

Database objects, like the spatio-temporal objects from previous sections, are to be stored *persistently*, that means the existence of an object is decoupled from the (operating system) process that created it. The objects are allocated in the so-called *persistent storage area* that is usually mapped onto secondary storage devices, such as hard disks. Programming with the persistent storage area differs from programming with main memory (volatile storage) area. For instance, to operate on a persistently stored object, the object must first be transferred (or *mapped*) from persistent storage area (the “database”) into main memory. In this thesis it is assumed that the basic unit of transfer is a so-called *page*, a fixed size unit of storage that is mostly sized to 4 Kbytes[1] or 8 Kbytes.

Update and access of persistent data occurs during *transactions*. A transaction is a sequence of database operations, which has the four well-known properties of atomicity, consistency, isolation, and durability (Haerder and Reuter, 1983). It is distinguished between *read-only* transactions and *update* transactions. Modern object storage systems offer the possibility to register function hooks on events pertaining to transactions (Biliris and Panagos, 1995; White and DeWitt, 1995; Lamb et al., 1991). Accordingly, an application is capable to execute functions on the following events: after begin of a transaction, after commit of a transaction, before abort of a transaction, before commit of a transaction, and before retry of a transaction. In this thesis, it is assumed that transaction management is performed by an object storage manager.

Secondary storage is partitioned into database clusters. In ObjectStore™, a cluster is a region on secondary storage of variable size that offers locality of reference. In this thesis, free space management is mostly left to the object storage system with one exception. The data structures to be described often have sub-parts of fixed size. This special case of free space management can be handled more efficiently. To this end, the following feature is exploited. When an ObjectStore™ array is allocated in persistent memory (starting in a new cluster) and the array has $\lfloor$pagesize/sizeof(element)$\rfloor$ elements, the array occupies a single secondary storage page. This way, one gets fine-grained control of (fixed-size) object placement.

Finally, the facility of so-called cross-database pointers is of importance. This ObjectStore™ facility can be used if objects in one database are to reference objects in another database. Although not used directly, this facility is the justification for the data partitioning argument given below.

5.2 Design Objectives

Several objectives guide the design of the storage architecture for the spatio-temporal data types. While some of them follow from the objectives for spatio-temporal data types on the user level (see previous chapter), others are new. The former comprise:

- Separation of a mesh, its simplices, and its vertices that are all linked by an n:m-relationship. Internal structures are affected, since they must reflect this separation adequately by resolving references or by assuring referential integrity.

- Object-oriented features and user-defined attributes for spatio-temporal objects. Internal structures are affected, since they must manage the link between user-defined data and spatio-temporal data.

- Support for temporal d-meshes. Internal structures are affected, since they must assure the restrictions defined on temporal meshes.

[1]1 Kbyte = 1024 bytes

- Transparent integration of pure-spatial and spatio-temporal data that are treated uniformly on the user-level. Internal structures are affected, since they remove this transparency such that proven spatial storage, access methods, and algorithms can be applied to pure-spatial data for reasons of efficiency.

The separation of a temporal mesh from its temporal simplices and its temporal vertices deserves a closer inspection. The storage structure should support the construction of several meshes from one or several groups of simplices and vertices. The separation objective has several implications. First, a (geometric) update operation on a temporal vertex must be propagated to the referring temporal simplices and meshes, hence to different structures. Second, such geometric updates invalidate *index structures* that must also be updated accordingly.

A further objective is to build a *dynamic* structure that allows for updates. The justification for this objective has been given above. At the same time, it is reasonable to assume that objects should be grouped according to a "nearness" criterion: objects near in space and time should be near on secondary storage. This makes sense, since spatial, temporal, and spatio-temporal *range queries* are considered prevalent in applications.

For a similar reason, the proposed storage structure should allow for explicit representation of certain *topological* information: the (time-varying) *neighbours* of a temporal mesh-simplex should be accessible fast. Being redundant, this information must be treated with care during updates, but it nevertheless proves useful in implementations for operations on temporal complexes, as will be demonstrated soon.

Since geometric objects are fairly large, there are two main options for the grouping on secondary storage. First, storage of a set of objects can be determined *locally*, which means for each object separately, such that each page contains only data belonging to one and the same object. Second, storage of a set of objects can be determined *globally* such that each page may contain data from several objects. While the former is advantageous if queries and operations pertain to a single object, for instance, intersecting a spatio-temporal object with a given plane, the latter is advantageous if queries are related to sets of objects, for instance, performing range queries over a set of objects. From this discussion, one can conclude that neither is superior to the other strategy of grouping. At the same time, the decision can well be done on the application level. Therefore, the storage structure should allow for a compromise of global versus local clustering, decidable by application developers.

5.3 Overview of the Storage Structure Components

The proposed system consists of an external and an internal level. While the former comprises the spatio-temporal data types within an object-oriented framework and has been the focus of the previous chapters, the latter comprises storage methods for the geometric part of the spatio-temporal objects. However, there must exist a "glue" that holds both levels together. This "glue"—also a component of the storage structure—is discussed in section 5.4.

The internal part of the storage structure consists of several sub-components. First, there is a point-manager component responsible for the management and the retrieval of

(Sub-)Component	Responsibilities
point-manager	clustering of point data; retrieving point data through: (1) range query; (2) object identity
object-header manager for *point*-objects	bookkeeping information; serves as an indirection to turn the points into relocatable objects
object-header manager for other temporal simplices	bookkeeping information

Table 5.1: Overview of the components of the internal storage structure and their responsibilities.

time-dependent point objects. Thereby, an application can allocate an arbitrary number of point-managers. These properties facilitate data partitioning. Second, for every temporal simplex data type (*point*, *segment*, *triangle*, *tetrahedron*), there is an object-header manager, which carries bookkeeping information about the objects (described below). Furthermore, the object-headers for *point* objects are used as an *indirection* during retrieval such that the *point* data becomes relocatable, necessary for clustering. Again, an application can allocate an arbitrary number of object-header managers, facilitating data partitioning. Concerning the logical storage hierarchy that is based on references, a reference graph is materialised for references "down" the hierarchy, as well as for references "up" the hierarchy. The latter is needed to check the validity of updates. Concerning the management of meshes, additional structures are used to maintain neighbourhood-relationships between simplices and to maintain connected components. The internal part of the storage structure is summarised in table 5.1.

5.4 Linking Spatio-Temporal Objects with Thematic Data

Representing, storing, and retrieving spatio-temporal data is in itself not sufficient for a spatio-temporal database system of a GIS. Moreover, it must also be possible to associate this data with user-defined, thematic data. Normally, the data model of the underlying DBMS is used for this purpose. In the case of an object-oriented DBMS, the new classes can participate in aggregations, associations, or inheritance and establish a connection between thematic and spatio-temporal data; however, the introduction of the new classes raises several questions. For instance, since mesh objects are *collections* of simplices, there is the question whether substitution polymorphism is supported. With this feature, an application can insert simplex objects into a mesh object that are in fact instances of a class *derived* from the particular simplex class (and obtain a reference to this derived class on request). A problem exists, since geometric and thematic data are separated internally for the reasons given above. By separating these parts, the usual mechanism for inheritance are thwarted. This section explores the problem of how to achieve and manage this association. First, it clarifies how thematic data can be embedded from a user‘s point of view. Then, it develops the structural aspects of the association. Finally, it develops the dynamic aspects of the association (creation, deletion, and update).

5.4.1 Thematic Data

Having defined the *stObject* class hierarchy within an object-oriented framework, an application can embed these classes in its own database schema. For instance, the *stObject* classes can participate in aggregations (in the sense of object-oriented modelling). Such connections with application-specific classes can already be seen as an association of thematic data with spatio-temporal data.

One elegant way of associating thematic data with a spatio-temporal object that is particularly supported is *inheriting* from the *stObject* class hierarchy. For example, an application wants to store thematic data along with every temporal point of a temporal surface and creates therefore a class derived from *point*, say *myPoint*, which carries the thematic attributes. Then, the application can construct *triangle* objects using *myPoint* objects and can in turn construct *triangleNet* objects using these *triangle* objects. Furthermore, the application expects that references to *myPoint* objects are returned by the system when it accesses the points of such a *triangleNet* object. This requirement has been termed "substitution polymorphism" requirement above. The remainder of the section addresses the problems of associating thematic data with spatio-temporal data through inheritance from the *stObject* class hierarchy.

5.4.2 Bridging the User-Level and the Internal Level

Within the object storage system ObjectStore^{TM} no low-level control of where attributes of classes are stored physically is available (all attributes of an object are stored together). Therefore, separating thematic attributes from geometric attributes, when the former are introduced through inheritance, can only be accomplished, if one part is "factored out" into a separate inheritance tree. At the same time both parts must be linked to remain accessible when given a spatio-temporal object.

In the above *myPoint*-example, the following constructor of *triangle* can be used to construct a *triangle* object:

triangle::new(Ref<point>, Ref<point>, Ref<point>)

Via polymorphism also references to *myPoint* objects may be passed as argument; however, the constructor is unaware of the potential types of its parameters and simply *copying* the *point*-part of the parameters results in a loss of the user-defined attributes: the *triangle* object would refer only to *point* objects, but not to *myPoint* objects. Likewise, although using a technique like a virtual clone-method would solve this problem, it is not applicable, since it would violate condition (1) of separating the geometry part from the thematic part of an object. Therefore, on creation of a *point* object, the geometric part is copied into the internal structure while establishing a bidirectional link between the geometric and the thematic part.

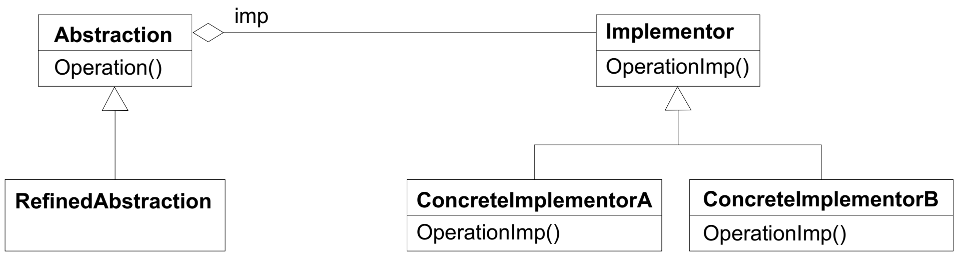


Figure 5.1: The structure of the "bridge" design pattern (taken from Gamma et al., 1995).

The design pattern "bridge" (Gamma et al., 1995) solves the problem of binding representations to the data types on the external level. Originally, the intent of the "bridge" design pattern is to decouple an abstraction from its implementation (Gamma et al., 1995). Its structure is depicted in figure 5.1. In the setting here, the abstraction mentioned in the pattern corresponds to the *stObject* class for spatio-temporal objects, derived classes of which carry the thematic part of the objects. The implementation mentioned in the pattern corresponds to the internal level responsible for the geometric part of the objects. Hence, the "bridge" separates both parts of a spatio-temporal object. At the same time, the bidirectional link offers a traversal path from the internal to the external part and vice versa. The internal part is well hidden from the external part and reduces dependencies among both. The "semantics" of the data types on the external level can be preserved, although this needs still to be demonstrated (see below). Furthermore, this pattern allows to change the representation of a given spatio-temporal object at runtime. Here, the "bridge" pattern offers an elegant solution.

The design pattern "bridge" is applied as follows. For each concrete class derived from *stObject* (*point*, *segment*, *triangle*, *tetrahedron*, *polyline*, *triangleNet*, *tetraNet*, and *Group*) there is one instantiation of the pattern. In contrast to the description of Gamma et al. (1995), the aggregation association is not only directed from the abstraction side to the implementation side, but also directed oppositely. The latter direction is needed for a traversal path for the internal structures to the thematic data part. Alternatively, this opposite link can be avoided, if on the implementation side any reference is towards the abstraction side. For example, the implementation object for a *triangle* object (on the implementation side) stores references to three *point* objects (on the abstraction side) rather than references to three implementation objects of *point* objects (on the implementation side). Although this approach would save storage space, it is not suggested here mainly for two reasons. First and less important, the alternative approach results in an additional dereference, which leads in turn to more disk accesses. Second, the implementation object for any spatio-temporal object is forced to have a persistent counterpart on the abstraction side of the bridge, even for the case that a spatio-temporal object is not intended to be linked to thematic data at all. Especially for smaller objects like *points* this would entail an intolerable overhead. Instead, for this case the counterpart objects on the abstraction side can be created transiently as needed, and this is described subsequently.

The dynamic aspects of the bridge can be described as follows. The implementation side of the bridge offers the same interface as does its counterpart on the abstraction side of the bridge. These interfaces have been described in the previous chapter. A call of a method on the latter side is forwarded to its counterpart on the former side by using the aggregation association. The construction of a spatio-temporal object is triggered by the user calling one of the constructors of one of the *stObject*-subclasses (on the user/external level). The constructor is responsible for constructing both sides of the bridge; however, the default constructor without arguments only constructs the abstraction side while setting the reference to the implementation side to null. The methods of such an object act like those for a spatio-temporal object with an empty lifetime. For the remaining constructors, the construction of the abstraction side can occur in two different ways, depending on the user‘s choice. First, the object on the abstraction side can be constructed *persistently* for storage in the database. Second, the object on the abstraction side can be constructed *transiently.* The proposed system can use this as a reasonable hint that the user of the *stObject* hierarchy wants or does not want permanent linkage to thematic data, respectively:

- One of the subclasses of *stObject* is instantiated persistently. The system interprets the creation such that a linkage to thematic data is intended and creates the "complete" bridge with the bidirectional link.
- One of the subclasses of *stObject* is instantiated transiently. The system interprets the creation such that a linkage to thematic data is not intended. It creates only the implementation side of the bridge persistently. The reference from this side to the abstraction side is set to null. There is a special mechanism to ensure that the full bridge "is there" when needed.

The mentioned mechanism must address two important issues. First, it must somehow enable persistent references to transient objects; indeed, this is necessary, since (a) such references must not be written to persistent storage; and (b) a read-only transaction is not allowed to manipulate these references directly. Second, the lifetime[2] of these transiently allocated objects must be managed. The first issue is solved through ObjectStore^{TM}‘s facility to manage persistent pointers to transient objects, as is needed here. Details can be found in (ObjectStore, 2003). The second issue is solved with the concept of a transaction. Since access to persistent data occurs always within a transaction, the lifetime of the transiently created objects is bound to the scope of a transaction. By establishing a transaction-hook these objects can be deallocated at transaction commit or abort. Whenever a reference to an object on the abstraction side is to be returned, an encapsulating method on the implementation side is called:

Ref<abstraction> *implementation*::get_abstraction()

The method implementation checks the current reference for equality to null. If this holds, it creates a transient abstraction object and establishes the persistent-to-transient reference within the object storage system.

[2]In the sense of creation and deletion of objects

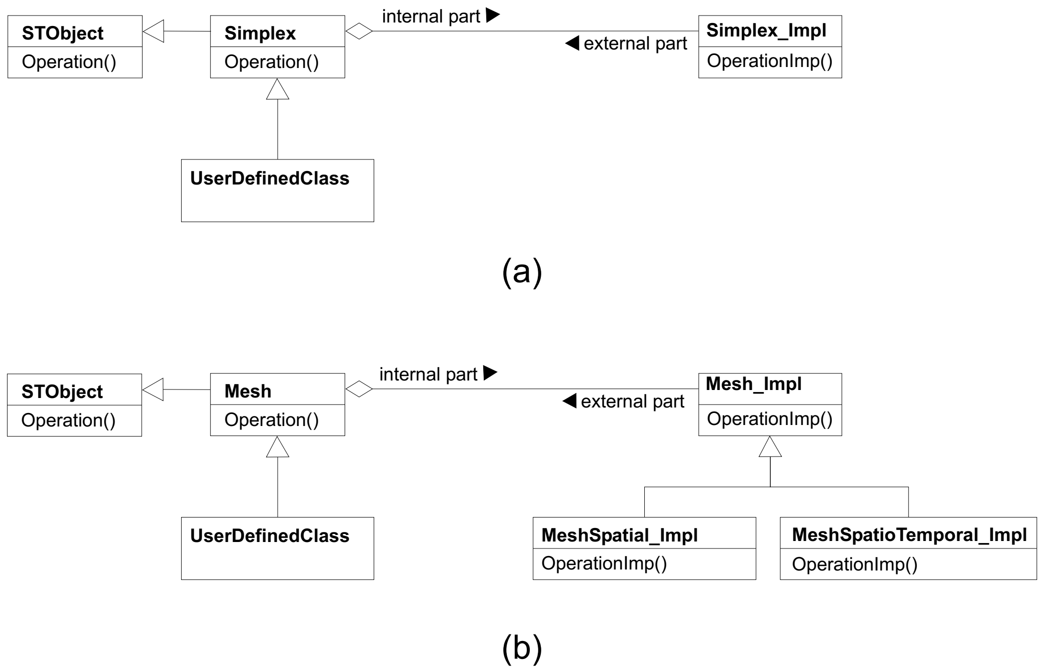


Figure 5.2: The application of the "bridge" design pattern for (a) the simplex classes and (b) for the mesh classes.

The application of the "bridge" pattern is analogous for all simplex classes and for all mesh classes. The application of the pattern for the simplex classes is shown in figure 5.2(a). It is worth noting that—for the simplex classes—there is no abstract base class on the implementation side of the bridge, since it entails a hidden pointer in the C++ representation[3] and, hence, demands more storage space than necessary. The different treatment for atemporal and temporal spatial data is realised in a further data structure to be described below. The application of the pattern for the mesh classes is shown in figure 5.2(b). In particular, the figure shows on the implementation side an abstract base class for two mesh representation classes, one for the case of a pure-spatial mesh and one for a temporal mesh. The former can be based on proven spatial data structures, algorithms, and access methods, while on the external level a unified interface avoids a proliferation of classes.

The proposed solution exploits further properties of the bridge pattern. First, both the abstraction side and the implementation side are extensible by subclassing and can vary independently. Thereby, the abstraction side is intended for application designers who create their application schemas using the *stObject* hierarchy. Second, it is possible to switch the implementation at runtime, while leaving the abstraction side almost unaffected. Third, objects on the implementation side of the bridge can be shared among each other and this interdependence can be managed on the implementation side. To sum up, the design pattern "bridge" fits the problem of binding representations to spatio-temporal objects well and many properties of it can be exploited. The remainder of the chapter covers the implementation

[3]This is the virtual function table pointer that is needed to resolve late bound methods.

side of the "bridge" pattern instantiations.

5.4.3 Managing Object References

The hierarchy of spatio-temporal objects is based on *references* (see also figure 4.1 on page 46): a temporal mesh aggregates temporal simplices, while a temporal simplex aggregates temporal points. To avoid dangling references, a mechanism for their management is built into the data structures. The aggregated objects exist independently from their aggregating objects, establishing an n:m-relationship among them: a single temporal point can be aggregated by an arbitrary number of temporal simplices, while a temporal simplex can be aggregated by an arbitrary number of temporal meshes. Hence, there are two different paths along the hierarchy of spatio-temporal objects. First, there is the path *mesh-simplex-point* "down" the hierarchy. Second, there is the path *point-simplex-mesh* "up" the hierarchy. Obviously, the former path must be materialised, as it describes the way that spatio-temporal objects are constructed; however, the latter path must be materialised as well, as it describes the way that updates are checked against restrictions. The following interface is used to manipulate the path "up" the hierarchy:

Method name	**Parameter types**	**Return type**
Class *Point_Impl*:		
insert_simplex_reference:	Ref<Simplex>	→ void
remove_simplex_reference:	Ref<Simplex>	→ void
Class *Simplex_Impl*:		
insert_mesh_reference:	Ref<Mesh>	→ void
remove_mesh_reference:	Ref<Mesh>	→ void

By materialising the path "up" the hierarchy of spatio-temporal objects, it becomes possible to manage the automatic deletion of the objects on the internal side. A deletion is triggered by the user calling the destructor of an object of the *stObject* class system (see "bridge"-discussion in section 5.4.2). Since an object cannot be accessed any longer, if no references to it exist, the object can then be deleted safely; however, such an object can be part of another object on the next higher level of the hierarchy. As mentioned above, the system materialises this relationship and it can thus decide, whether the object is part of another one. If it is not, the object can be deleted. If it is, whether deletion occurs or not depends on the parameter passed to the destroy-operation. Consequently, the deletion-operation "*o*->destroy(*option*)" of a spatio-temporal object *o* is performed as follows: (1) the reference *o*->*internal*->*external* is set to null; this avoids an endless loop that could otherwise occur, since the destructor of an implementation object calls the destructor of its abstraction object; (2) the destructor of *o*->*internal*->destroy(*option*) is called; if the destructor raises an exception, the reference *o*->*internal*->*external* is restored and an exception is raised; (3) otherwise cleanup of the user-defined thematic data (user‘s responsibility). The call of the destruction operation *o*->*internal*->destroy(*option*) in (2) is managed in the following way, depending on the particular type and parameter *option* (see also algorithms 5.4.1, 5.4.2, and 5.4.3). For a point object (algorithm 5.4.1) there are the options: (1) DEFAULT (lines 2–5) for the default behaviour of not deleting a point in case there are references to it and to throw an exception in that case; (2) TRIGGER_CASCADE (lines 6–8) for triggering the deletion of any referring simplex object, which in turn removes the simplex object from any referring mesh object.

Algorithm 5.4.1 point_impl::destroy(*option*)

```
1: switch option
2:    case DEFAULT :
3:       if this->external=null and this->simplex_references=∅ then delete this
4:       else throw exception
5:    end case
6:    case TRIGGER_CASCADE :
7:       for all s in this->simplex_references do s->destroy(TRIGGER_REMOVE)
8:    end case
9: end switch
10: if this->external ≠null then delete this->external
11: delete this
```

For a simplex object (algorithm 5.4.2) there are the options: (1) DEFAULT (lines 2–5) for the default behaviour of not deleting a simplex in case there are references to it and to throw an exception in that case; (2) TRIGGER_REMOVE (lines 6–8) for triggering the removal of the simplex from any referring mesh object. Furthermore, management of the path "up" the hierarchy occurs in line 11.

Algorithm 5.4.2 simplex_impl::destroy(*option*)

```
1: switch option
2:    case DEFAULT :
3:       if this->external=null and this->mesh_references=∅ then delete this
4:       else throw exception
5:    end case
6:    case TRIGGER_REMOVE :
7:       for all m in this->mesh_references do m->remove(this)
8:    end case
9: end switch
10: if this->external ≠null then delete this->external
11: for all points p in this do p->remove_simplex_reference(this)
12: delete this
```

For a mesh object (algorithm 5.4.3), no options are needed and the necessary steps are to manage the path "up" the hierarchy in line 2.

Algorithm 5.4.3 mesh_impl::destroy()

```
1: if this->external ≠null then delete this->external
2: for all simplices s in this->simplices() do s->remove_mesh_reference(this)
3: delete this
```

5.5 Indexing Spatio-Temporal Objects

The purpose of indexing is to speed up searches within a collection of database objects. Usually, an object to be maintained in an index is associated with a single key. Then, an index efficiently supports queries for those objects whose keys match a given query key (*exact match query*), and queries for those objects whose keys lie in a given range of keys (*range query*). Focusing on spatio-temporal objects, one can observe that using indexes helps solving the following important problems: (1) speed-up of range queries on spatio-temporal objects; (2) solving related search problems—for example queries based on a distance predicate—efficiently without having to pre-process the complete data set; (3) speed-up of updates[4] on temporal meshes that must check the restrictions mentioned in the previous chapter; (4) grouping the geometric part of spatio-temporal objects on secondary storage in such a way that retrieval operations keep the number of disk accesses low. For these reasons, this section looks at the problem of indexing of spatio-temporal objects in more detail.

The remainder of the section is organised as follows. The next paragraph reviews current research work on index structures for spatio-temporal data and comes to the conclusion to adapt the well-known R*-tree index structure for spatial data. Then, the following paragraph introduces this index structure in more detail. Finally, the last paragraph centres on how to adapt the R*-tree for spatio-temporal data.

5.5.1 Related Work on Spatio-Temporal Indexing

There has been extensive research on the problem of indexing spatio-temporal objects over the last years. A recent survey of this field of research have been given by Nascimento et al. (1999) and Pasquale et al. (2003). This section describes first several straightforward approaches and proceeds then to more novel indexing techniques.

One straightforward approach is based on a two-step process, falling back on available spatial and non-spatial indexes. The strategy is to perform a query separately for the temporal part and for the spatial part of a spatio-temporal range query, and to combine the results appropriately. This strategy has been introduced as a "simple scheme" of spatio-temporal indexing by Theodoridis et al. (1996, p. 443). Moreover, in the context of the ObjectStore$^{\text{TM}}$-based GeoToolKit system, this strategy has been described by Balovnev and Breunig (1997, p. 113). It can be summarised as follows:

- perform a temporal range query based on the object storage query facility to obtain a collection R_t of all objects within the given *temporal* range
- if the size of R_t is small: perform a spatial range query by iterating through R_t and testing each object for containment within the given *spatial* range, obtaining the final result.
- else if the size of R_t is large:

[4]Usually, it is assumed that indexing slows down updates, while it speeds up queries; however, here it is necessary to check the mentioned restrictions before processing an update. These checks benefit from the presence of indexes.

– perform a spatial range query using a spatial index to obtain all objects within the given spatial range, obtaining the collection R_s;

– Output those objects as result, which are contained both in R_t and R_s.

However, Balovnev and Breunig (1997, p. 113) expected that in most cases using multi-dimensional indexes would yield better performance.

A further straightforward approach was suggested by Xu et al. (1990) who altered the R-tree spatial index. In the variant tree structure, termed RT-tree, each node is extended to hold a temporal interval covering the intervals of its descendants in analogy to the minimum bounding box; however, the insertion process of the RT-tree is guided mainly by the spatial part of the nodes, although on node overflow, the authors propose a node splitting strategy based on a time interval preference (Xu et al., 1990, p. 1046). Notwithstanding, the overall structure seems to arrange data more according to space than to both space and time (see also Nascimento et al., 1999).

The final straightforward approach to be described is that of using an available multi-dimensional index structure, while treating time as a further dimension. Theodoridis et al. (1996) termed this approach the "unified scheme" for their index structure. The idea is to approximate a two-dimensional spatio-temporal object by a three-dimensional minimum bounding box, which describes the extent of the object's trajectory for the spatial dimensions and the lifetime of the object for the time dimension. The bounding boxes are in turn indexed and queried using a three-dimensional R-tree "as-is".

Several index structures for spatio-temporal objects have been suggested that are based on the concept of node sharing among several trees, yielding a directed acyclic graph. The idea is to maintain an index tree for every time-step, but to "re-use" nodes that did not change from one time-step to the next. One tree, valid for a given temporal interval, contains the whole indexed data set, whereas further trees, valid for later temporal intervals, only hold the changes to the data set and refer to nodes of other trees for the unchanged data. For example, the MR-tree of Xu et al. (1990), the HR-tree of Nascimento and Silva (1998) and the Overlapping Linear Quadtree of Tzouramanis et al. (2000) try to exploit this strategy. Obviously, these approaches are most promising when a large part of the indexed data set remains unchanged from one time-step to the next; however, as has been mentioned in chapter 3, in the target applications for this work it is expected that data changes continuously, and therefore "invalidating" many index nodes from one time-step to the next. This fact renders the approaches based on node sharing not as effective as in their own target applications of discretely moving points.

In contrast, much work has been targeted at the application domain of continuously moving points. The work divides into two different indexing problems (Saltenis et al., 2000): (1) indexing the current and anticipated future positions of moving objects; and (2) indexing the histories of the positions of moving objects. Among the work that addresses the former problem is that of Saltenis et al. (2000) who extend the R-tree to a time-parameterised R-tree (TPR-tree). This tree stores moving objects of a constant velocity vector within the TPR-tree nodes. Interestingly, the bounding rectangles of these nodes are functions of time that must be adjusted when the trajectory of an object changes; however, more complex trajectories (e.g. piecewise-linear trajectories) cannot be indexed. Closely related to the

TPR-tree is the work of (Procopiuc et al., 2002). The authors propose a spatio-temporal self-adjusting R-tree (STAR-tree) for indexing the trajectories of moving points. The tree maintains auxiliary information and updates itself locally to ensure that queries are answered efficiently (Procopiuc et al., 2002).

A different approach is taken, if duality transforms of lines in space are exploited. Given that an object moves on a straight line segment in space (bounded by a temporal interval of validity), the trajectory of such an object can be transformed to a point in dual space. The points in dual space can then be indexed by spatial indexes. One such approach has been described by Agarwal et al. (2000). Moreover, among this approach is the work of Papadopoulos et al. (2002). The authors propose an indexing scheme that makes use of two different dual transforms, where one treats the case of “slow” objects. Two-dimensional movement (movement in the xy-plane) is indexed by two separate R*-trees (Beckmann et al., 1990), one for each dimension and are responsible of indexing the points in dual space. Range queries must be transformed as well, yielding so-called simplex range queries that the authors approximate by orthogonal range queries suitable for the R*-tree.

Importantly, the previously mentioned approaches pertain to indexing the currently known movement of a point object, mostly represented as a constant velocity vector. Since the history of the movements are not stored, as is necessary for this work, these indexes cannot be used here.

For a similar reason, the technique of making the indexed data *persist* with so-called *partially persistent data structures* (Driscoll et al., 1989) must be ruled out. A partially persistent data structure maintains different versions of a structure caused by updates, and it allows to *access* versions earlier than the current version, and it allows to access and to *update* the current version. Although the history of an object‘s trajectory can be accessed with this technique, partially persistent data structures allow per definition updates only to the current state of an object‘s trajectory. A partially persistent data structure for spatio-temporal objects has been presented by Kollios et al. (2001) and Hadjieleftheriou et al. (2002), who introduce the partially persistent R-tree (PPR-tree). Furthermore, the MV3R-tree (Tao and Papadias, 2001) is also targeted at the moving objects application domain and combines ideas from multi-version B-trees (Becker et al., 1996) and 3D-R-trees. The latter treats time as the third dimension. The proposed index structure applies two separate indexes: a multi-version R-tree (MVR-tree, a partially persistent data structure) and a 3D-R-tree that together form the MV3R-tree index. Both are connected in that the 3D-R-tree is used to index the leaves of the MVR-tree. The rationale behind the approach is that the MVR-tree is expected to be more efficient for snapshot and “short“ temporal interval range queries, while the R-tree is expected to be more efficient for “large” temporal interval queries.

A further technique that is often applied is that of trajectory splitting. The rationale behind this technique is that approximating the (complete) trajectory of a spatio-temporal object by (d+1)-dimensional bounding boxes is poor (Theodoridis et al., 1998). Splitting the trajectory of an object at a set of instants improves the approximation in general, although at the cost of a greater number of objects that are to be maintained by an index. Several split algorithms have been given by Hadjieleftheriou et al. (2002). On the contrary, Pfoser et al. (2000) model the trajectories of two-dimensional moving points as poly-lines (piecewise-linear) in three-dimensional space and split them into their constituent segments. These trajectories are then indexed by the spatio-temporal R-tree (STR-tree), introduced by the authors. The

STR-tree is a variant of the R-tree and has the additional property of preserving trajectories when updating the index: the index aims at placing the bounding box for a segment “near” the predecessor-segment of the segment‘s trajectory. Originally, the R-tree aims at preserving spatial nearness.

To sum up, many index structures have been proposed in literature. However, most of them are targeted at the application domain of moving objects in a two-dimensional space in an on-line scenario: the trajectories of objects are updated at certain intervals, while the history of trajectories is either of no interest or cannot be updated. The index structures that meet the requirement of the application domains for this thesis are the three straightforward approaches described above and the STR-tree of Pfoser et al. (2000) for moving point objects. It is generally assumed that the former ones perform poorly. The latter one focuses on moving point objects with trajectory preservation as described above. However, trajectory preservation is not preferable for object-level indexing; deletion of objects has not been described by Pfoser et al. (2000); retrieval of objects depends on continuous movement (Pfoser et al., 2000, sect. 4.1), such that discontinuities and temporal “gaps” are ruled out. For these reasons, the next section describes an indexing scheme for spatio-temporal objects.

5.5.2 Adapting R-Trees for Spatio-Temporal Objects

This section presents a slight modification of the R-tree index structure for spatio-temporal objects. It accounts for the splitting of object trajectories, while also enabling retrieval of trajectories of objects with discontinuous movement. As it is a direct modification of the R-tree structure, it allows for the update (change, extension, reduction, etc.) of any part of an object‘s trajectory. Since the focus is on grouping of temporal points and on indexing of a (single) temporal mesh, trajectory preservation as in Pfoser et al. (2000) is not preferred. Before introducing the modified structure, the R-tree index structure is briefly reviewed.

The R-Tree for Spatial Indexing

An n-dimensional R-tree (Guttman, 1984) organises sets of objects with respect to their *keys*, which are n-dimensional hyper-rectangles, orthogonal to the coordinate axes. It is similar in nature to a B-tree (Bayer and McCreight, 1972; Comer, 1979; Knuth, 1973). It supports efficient processing of orthogonal range queries, which return references to those objects, the keys of which intersect the query hyper-rectangle. The R-tree is a dynamic data structure in that it allows to insert into and to remove objects from the set of indexed objects without the need for a global re-organisation.

The R-tree is a height-balanced tree that stores its indexed hyper-rectangles only in its leaves. The inner nodes of the tree, also called *directory nodes*, contain a number of entries of the form

(Ref<child>, hyper-rectangle)

where Ref<child> refers to a child node of the inner node and “hyper-rectangle” meets the condition that it is exactly the minimum bounding hyper-rectangle of the set of hyper-rectangles contained in the sub-tree rooted at Ref<child>. A leaf node has entries of the form

$$\text{(Ref<object>, hyper-rectangle)}$$

where Ref<object> references the indexed object and "hyper-rectangle" is the key for the object. The nodes have a maximum number M of entries. The nodes of an R-tree have the following properties (Guttman, 1984):

- If the root of the tree is not a leaf, then it has at least two child nodes.
- Each inner node, except the root, has between m and M child nodes ($2 \leq m \leq M/2$).
- Each leaf has between m and M entries.
- All leaves are on the same tree-level.

When an R-tree is built via insert- and remove-operations, the structure is guided by the following heuristic: *Minimise the* content[5] *of hyper-rectangles in the directory nodes.*

One of the numerous proposed R-tree variants is the R*-tree (Beckmann et al., 1990). The authors improved the original R-tree by two strategies that influence upon tree structure: (1) they made use of different heuristics (reduction of the content as well as the margin and overlap of directory hyper-rectangles); (2) they introduced a forced re-insert strategy in case a node overflows during an insert-operation.

It is important to note that unbounded temporal intervals (those that have $-\infty$ (∞) as lower (upper) boundary) are accounted for by the R-tree index: "Alternatively I_i [a key‘s interval in one dimension] may have one or both endpoints equal to infinity [...]" (Guttman, 1984, p. 48).

An R-Tree-Variant for Spatio-Temporal Object Indexing

A four-dimensional R-tree can be used "as-is" to index a collection of spatio-temporal objects, but doing so results in poor retrieval performance. This fact has been observed by Theodoridis et al. (1998). The authors of that work refer to the low discriminating factor that degrades the R-tree‘s performance (see figure 5.3). Indeed, using 4D-R-trees without further ado means to approximate a spatio-temporal object (1) spatially by the three-dimensional minimum bounding box of the object‘s trajectory and (2) temporally by the hull of the object‘s lifetime. Together, the approximation forms a four-dimensional minimum bounding box that can be maintained by a four-dimensional R-tree.

5.5.1. DEFINITION. (*4D bounding box*) A *four-dimensional (4D) bounding box* b is a four-tuple of intervals $b = (I_x, I_y, I_z, I_t)$, where I_x, I_y, and I_z are closed intervals in $\mathbb{I}$ and denote the spatial extent of b, whereas $I_t \in \mathbb{I}$ denotes the temporal extent of b.

[5]Content is the generalisation of volume to higher dimensions.

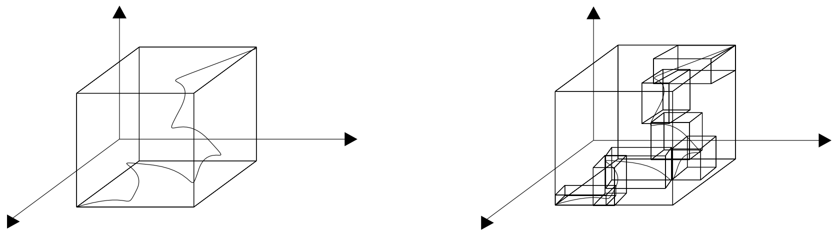


Figure 5.3: The low discriminating factor that Theodoridis et al. (1998) have referred to. (Note: time dimension is not shown.) The figure shows the trajectory of a temporal point that is approximated by a single 4D bounding box (left) and by several 4D bounding boxes (right) by splitting the trajectory. Observe also, that the left scheme does not support snapshot queries ("retrieve point location at instant *t*").

5.5.2. Definition. (*class* boundBox4D) Class *boundBox4D* models a 4D bounding box. Its structure is defined as follows: *boundBox4D* = [bb: *boundBox*, time: *interval*].

Figure 5.3 shows the approximation of a spatio-temporal object and indicates the low discriminating factor that Theodoridis et al. (1998) have referred to. The negative effect is twofold. First, the low discriminating factor causes an unnecessarily high overlap of the maintained bounding boxes and, hence, a negative influence on the structure of the R-tree. Second, the low discriminating factor causes many false hits in range queries. Here, an object is called a false hit for a given query predicate, if its approximation (the 4D bounding box) meets the query predicate, whereas the object itself fails to meet the query predicate. One can conclude that the R-tree query-performance is poor, since the worse the approximation, the higher is the number of false hits in queries. Besides the low discriminating factor, effective use of a four-dimensional R-tree "as-is" is hindered by a further disadvantage. By approximating a spatio-temporal object by a single four-dimensional bounding box, an R-tree is ineffective with respect to snapshot queries (queries of the form *Show the object at instant t*, compare figure 5.3). To sum up, approximating a spatio-temporal object by a single four-dimensional bounding box results in poor index performance and does not support snapshot queries. These aspects lead to the conclusion to refine object approximation and to adapt the R-tree index structure to this end.

By allowing refinement of object approximation, the R-tree index structure is changed in the following ways. First, a spatio-temporal object is approximated by several four-dimensional bounding boxes to improve the discriminating factor, as well as the support for snapshot queries. To this end, a 1:n-relationship is established between objects and keys, in contrast to the "original" R-tree that establishes a 1:1-relationship between objects and keys. Second, the structure of an R-tree leaf entry is extended to store pointers that link all those keys that belong to a single object. Third, the R-tree algorithms are changed to account for the 1:n-relationship between objects and keys and to account for the maintenance of the new leaf entry pointers.

The first aspect of a 1:n-relationship between objects and keys raises the following issues. Approximating a spatio-temporal object by several four-dimensional bounding boxes (the keys) implies splitting the object‘s lifetime (and thereby the object‘s trajectory) into several pieces. This establishes a 1:n-relationship between objects and keys to be maintained by the index. Splitting can be performed in any reasonable way; however, the following structural requirements must be met: (1) the time intervals of the resulting 4D bounding boxes must intersect only at their endpoints (no overlap); (2) the union of the time intervals of the resulting 4D bounding boxes must be equal to the object‘s lifetime; (3) each resulting four-dimensional bounding box must (spatially) be the object‘s trajectory minimum bounding box, where the trajectory is limited to the time interval of the four-dimensional bounding box. More formal, the concept of a split leads to the following definition.

5.5.3. Definition. (*Split*) Given a spatio-temporal object o, a *split* of o is given by a finite set S of instants ($S \subset \mathbb{T}$) with the following properties:

- for all $s \in S : s \in \text{closure}(o.\text{lifetime})$
- $\partial\ o.\text{lifetime} \subseteq S$

Then, a split S of a spatio-temporal object o induces a set of 4D bounding boxes that together form the keys of o: keys(o, S) denotes the set of *boundBox4D* objects $[bb, time]$. Here, bb is the minimum bounding box of o‘s trajectory limited to the temporal interval $time$. The time intervals $time$ of these *boundBox4D* objects are formed each by two consecutive instants in S, with the exception of those intervals that lie outside the lifetime of o, that means without intervals of the form $< t_i, t_{i+1} >$, where t_i is the upper boundary and t_{i+1} is the lower boundary of intervals in o.lifetime.

The structure of the tree nodes are as follows. Inner nodes remain unchanged with respect to an “original” four-dimensional R-tree. However, the leaf nodes have a new structure (compare also figure 5.4). Each entry in a leaf is of the form:

multikey_entry = [key: *boundBox4D*, value: Ref<obj-type>,
prev: Ref<*multikey_entry*>, next: Ref<*multikey_entry*>]

That means, the entries for a key-sequence of an object are linked bidirectional. The links are advantageous for the remove-operation and for querying, as shown below.

To account for the 1:n-relationship and the links between leaf node entries, several R-tree algorithms must be changed. The principal changes are as follows.

- Insert(Ref<obj-type> obj, *keySequence* keys): The operation is used to insert an object, indexed by a key-sequence. The operation iterates over each element k of the key-sequence “keys” and calls the “original” R-tree “insert”-operation on the (obj, k)-pair. There is an auxiliary function that allocates a leaf entry (a *multikey_entry* object) that is also responsible for adjusting the links.
- Remove(Ref<obj-type> obj, *point4D* hint): The operation is used to remove an object and all its keys from the index. The second argument to the function is a *hint* for

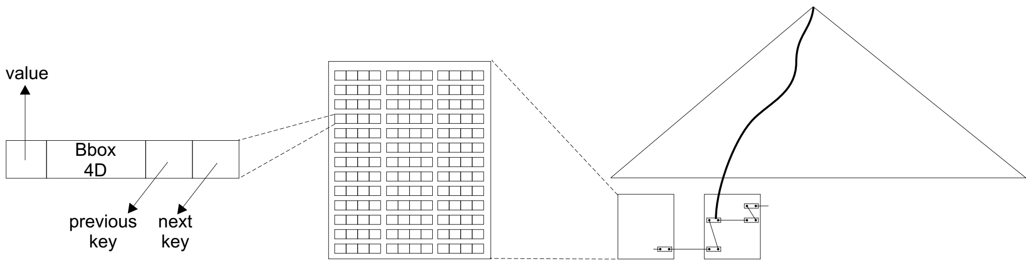


Figure 5.4: Structure of the 4D multi-key R-tree. The left part shows an entry in a tree‘s leaf node. The middle part shows schematically a leaf node that is mapped onto a secondary storage page. The right part illustrates a remove operation that is performed by searching the tree with a *hint* (an $(x, y, z, t) - tuple$ and the point‘s object-*id*) and then following the key-pointers to remove the whole key-sequence.

where to find one of the entries for “obj”. Having found one of the entries, the links are used to find the remaining entries for “obj”. The “original” R-tree remove-operation is called upon each such entry. However, care must be taken with the “original” sub-procedure “CondenseTree” (Guttman, 1984). This procedure is called by “remove” and serves two purposes: first, to propagate node elimination as needed and, second, to adjust directory hyper-rectangles. The problem with the new remove-operation is that “CondenseTree” re-organises leaf entries, if there is an underflow of leaf nodes after a removal, thereby potentially invalidating links of the key-sequence. A “safe” way of calling “CondenseTree” during removal is as follows. A removed entry is copied to a temporary location and the links are adjusted accordingly. Then, “CondenseTree” is called that can safely redistribute leaf entries. Then, the copied entry is deleted, adjusting the links accordingly.

- Search(*boundBox4D* query_box): This operation is a spatio-temporal range query that retrieves those objects whose keys intersect the query_box. Although the processing is not changed over the original procedure, it is emphasised that the return type is List<[Ref<obj-type>, *boundBox4D*]>.

To sum up, this section introduced a slightly modified R-tree index structure for the indexing of spatio-temporal objects. The main characteristics are the 1:n-relationship between objects and keys and the mechanism for their maintenance that is achieved by linking the key-sequence. The latter characteristic facilitates a fast removal-operation and a fast retrieval of the key-sequence with discontinuous object movement. This is in contrast to the work of Pfoser et al. (2000) who assume that two consecutive trajectory-pieces have a common end-point and exploit this continuity in their algorithms; however, trajectories with discontinuities and temporal “gaps” cannot be supported in this approach. In the following, the modified R-tree structure is used for indexing the simplices of a temporal mesh, as well as for the grouping of the geometric part (point objects) of spatio-temporal objects on secondary storage.

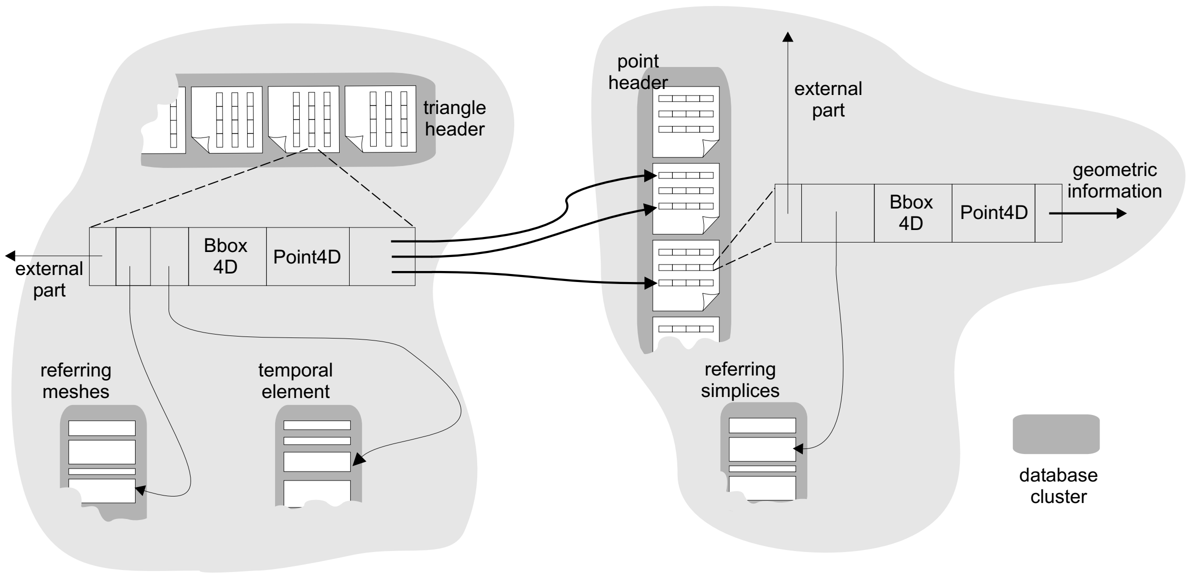


Figure 5.5: Organisation of the different parts of point and simplex headers within the database. Light shaded regions indicate simplex manager responsibility (left) and point manager responsibility (right). Dark shaded regions denote database clusters.

5.6 Storing Temporal Simplices

The task of storing temporal simplices comprises three parts. First, there is the task of storing temporal point-objects that is handled by the data structure “point manager”. Second, there are the tasks of storing temporal segment-, triangle-, and tetrahedron-objects that are handled each by a separate data structure, but can be described uniformly in the following, referred to as data structure “simplex manager”. Third, there is the task of managing bookkeeping information for the simplex objects handled by object headers.

5.6.1 Object Headers

A *point* or simplex object has associated with it an object header that holds bookkeeping information about the object. The owner of a point header is a point manager, and the owner of a simplex header is a simplex manager (described below). The clustering of the bookkeeping information on secondary storage is important, as it is often accessed in retrieval. Put simply, clustering aims at storing data together that is accessed together. Therefore, the information within the headers is split into several clusters, as follows (see also figure 5.5).

- **Point header** (fig. 5.5, right): One cluster contains pages (of secondary storage) that store 5-tuples, where each tuple consists of: (1) a reference to the external part of the object; (2) a reference to the set of referring simplices; (3) its 4D bounding box; (4) a

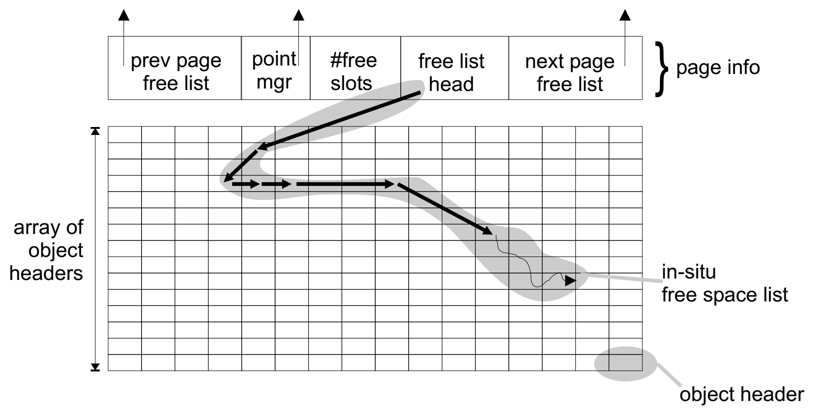


Figure 5.6: Page layout for object headers. The first part of a page contains bookkeeping information and the pointers for the list of pages that have free space. The larger second part contains the slots for the fixed-size part of an object header. These slots are also used for the page‘s free space list.

hint for the indexes (*point4D*); (5) a reference to the geometric information. A second cluster stores the set of referring simplices for each point.

- **Simplex header** (fig. 5.5, left): One cluster contains pages (of secondary storage) that store 6-tuples, where each tuple consists of: (1) a reference to the external part of the object; (2) a reference to the set of referring meshes; (3) a reference to its temporal element of validity; (4) its 4D bounding box; (5) a hint for the indexes; (6) references to the (fixed number of) vertices. A second cluster stores the temporal elements of validity for each simplex. A third cluster stores the set of referring meshes.

Since the environment is dynamic, objects that once have been created can be deleted at any time, leaving a “gap” in persistent memory. Then the problem of free space management must be addressed: where to allocate a newly created object. Here, the problem is in parts a simple sub-problem, as the entries for point headers and simplex headers are of fixed size and the first free “gap” of memory is adequate for allocation of a newly created header. Therefore, a mechanism should be realised that exploits these properties. The situation is different with the remaining information (temporal element of validity, sets of referring simplices/meshes), as it varies in size. Therefore, falling back to the memory management of the object storage system is an effective solution.

Conversely, the former (simpler) problem of allocating and de-allocating the fixed-size object headers is solved in an efficient special way. Furthermore, having control over the placement of object headers offers opportunities for *compaction*[6] and *clustering* of object headers in terms of access patterns. The solution organises free space in a way that can be described as a page-wise in-situ free space list. The layout of a page for this purpose is depicted in figure 5.6. A linked list of pages contains all those pages that have free slots. The

[6]Note that information about references to object headers for points and simplices is available and that thereby the headers are relocatable.

header of a page p contains pointers to the previous and the next page with free slots, an integer for the number of free slots in p, and a pointer to the head of the in-situ free space list within p. The list of pages with free slots is owned by the point manager and simplex manager for points and 1-, 2-, 3-simplices, respectively. Allocations and de-allocations of slots occurs only through these managers. The free slots within a single page are organised as an in-situ stack. On allocation of a new object header, the first free slot in the first page of the free space list is chosen and the slot is popped off the in-situ stack. If the stack is empty afterwards, the page is removed from the free-space list. On de-allocation of an object header in page p, the respective item is put on the top of the stack of p. If afterwards, every slot in p is empty (i.e. not allocated to an object header) the page is removed from the free space list and deleted from secondary storage.

5.6.2 Point Managers

The task of a point manager is to cluster parts of temporal points on secondary storage. The architecture of point managers has the following important properties:

- **Support of object-wise data partitioning.** Applications can create an arbitrary number of point managers that can reside in the same or in different database files.
- **R-Tree-driven clustering.** Internally, the adapted R-tree organises point data onto secondary storage pages.

A point manager allocates five database clusters: one cluster contains the 4D R-tree that is used to organise the clustering and four clusters for the point headers. A point manager is created and accessed via the following interface:

Method name	Parameter types	Return type
new:	void	$\rightarrow$ void
insert:	Set<[*interval*, *pl_curve*]>	$\rightarrow$ Ref<*point_impl*>
remove:	Ref<*point_impl*>	$\rightarrow$ void
retrieve:	*point_impl* $\times$ *boundBox4D* $\times$ Ref<*point_cache*>	$\rightarrow$ void

The available methods comprise construction, update, and retrieval. In brief, the constructor creates a point manager and allocates the above mentioned data structures for the point headers. Method “insert” is used to populate the point manager with point data. The method is called by the constructors of the *point* class. Method “remove” removes its argument from the point manager. Finally, method “retrieve”, used to access *point* objects for a given spatial and temporal range, fills a cache that maps *point* objects to *point3D* objects. This method exploits the clustering of *point* objects by filling the cache with data residing on the same page as the demanded data. More details are given in the next section.

The clustering is organised by the 4D R-tree that has been described in section 5.5.2. The following issues must be addressed: (1) the splitting strategy (definition 5.5.3) for temporal points; (2) the modification of the R-tree leaf entries. First, the split S of a point object p to be stored within a point manager comprises the boundary instants of the lifetime of the point and

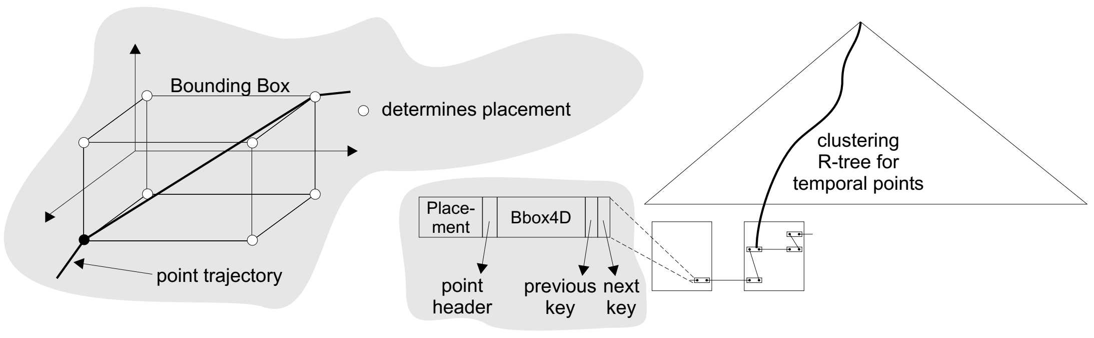


Figure 5.7: Clustering R-tree leaf entry. Every piece of the piecewise linear point-trajectory is stored in its own leaf entry. Together with the temporal interval (within the 4D bounding box), "placement" specifies the point coordinates, which need not be stored explicitly (see also Pfoser et al., 2000).

the support points of its trajectory (see page 31). These instants are to be derived from the Set<[*interval*, *pl_curve*]> argument to "insert"[7]. Then, each key in $keys(p, S)$ corresponds to a trajectory segment. Second, the leaf entries of the R-tree are modified slightly (see figure 5.7). An additional data field "placement" encodes the movement of the point along the trajectory segment. If the corners of the bounding box are numbered, the movement can be encoded by a single number n (see also Pfoser et al., 2000): at the lower boundary of the temporal interval (within the 4D bounding box) the point is located at corner n, and at the upper boundary it is located at the corner opposite to n. For each instant within the interval, the location of the point is derived by linear interpolation between these two locations (as defined above). Finally, the data field "value" in a leaf entry is assigned to the header of the point.

The process of inserting a *point* object can be sketched as follows. Firstly, a point header is allocated as has been described above. Then, the modified R-tree entries are computed. Each key of the split is inserted into the R-tree of the point's point manager. Finally, the reference to the geometric data of the inserted point is set. The process of deleting a *point* object comprises the two steps of deallocating the point header and the removal of every R-tree entry. In particular, the second step utilises the "previous/next"-references in an R-tree entry to iterate over those entries.

5.6.3 Transient Point Caches

Access to a point for an instant in time is likely to be followed by access to a point that is near in time or space. At the same time, this locality of reference is reflected by the clustering R-tree for temporal points. To exploit this locality of reference, a transient point cache is used during the retrieval of temporal points. Once a requested [point, instant]-pair has been retrieved, it is held in the cache such that a further request to the point for the same instant

[7]The *pl_curve* is defined in section 4.5.4

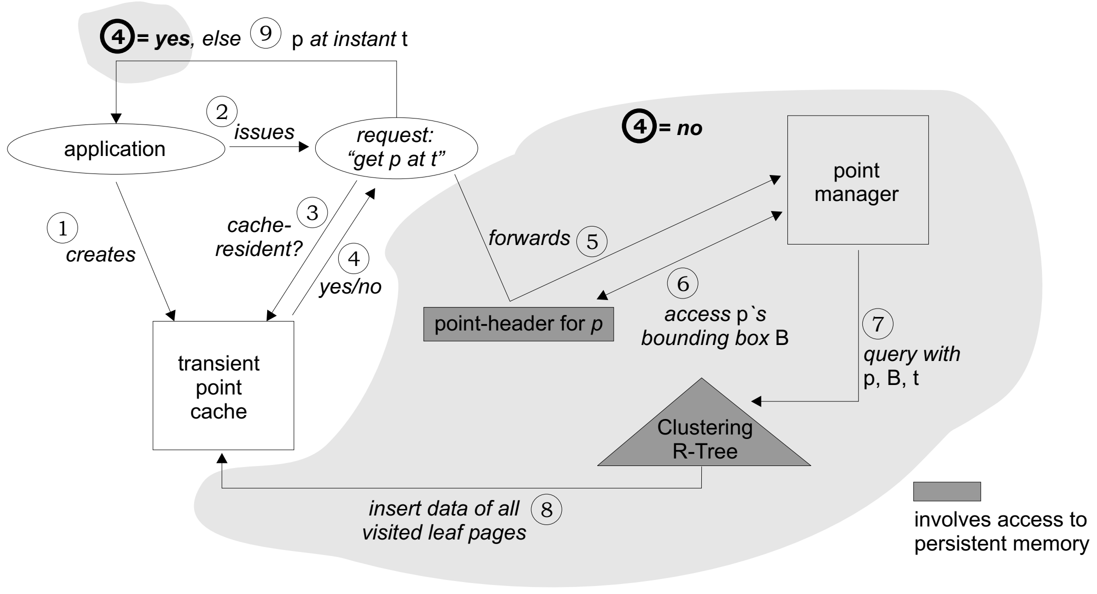


Figure 5.8: Processing of a point request.

or for a "near" instant can be processed fast. Thereby, "near" point-data is pre-fetched such that accessing these data does not necessitate an access to secondary storage.

An instance of a transient point cache is created and deleted by the access algorithms to be described subsequently. The instance can then be passed as argument to the retrieve-method of a point manager. The processing of a request for a temporal point is depicted in figure 5.8. The processing of a point request (get 3D point coordinates for a point object at an instant in time) is summarised as follows. After creation of a *point_cache* instance, the instance can be passed as argument to the point manager‘s retrieve-method. When a request for a snapshot of a point p at instant t is issued, the cache is first checked for main-memory residency of the requested point data. In the positive case, the request is serviced. Otherwise, the request is forwarded to p‘s responsible point manager. The point manager accesses the point‘s bounding box (stored in the point‘s header) to search the clustering persistent R-tree for the requested *point_impl*-reference. The R-tree search method is modified such that the point data in each leaf node that has been visited in this process, is inserted into the point cache. Eventually, the requested point is found, its data is stored in the cache and can be returned.

A transient point cache has a fixed size capacity (specified at creation). If the number of elements in the cache exceeds a threshold value, a least-recently-used replacement (LRU) policy is followed. The rationale behind this approach is as follows. Assuming that accesses follow the locality of reference principle, points in the cache that have not been accessed for a long period, have left the focus of interest. Therefore, a point cache maintains an LRU-list of its points. On insertion of a point, the point is stored at the end of the LRU-list. Points are removed from the head of the list. If a point is requested (accessed again), it is put at the end of the LRU-list.

The transient point cache can be realised as a hash-table, where the keys are pointers to *point_impl*-objects. The value for a key has the following structure. Class *rt-entry* denotes the class for entries in an R-tree leaf node.

[value: Ref<*rt-entry*>, LRU-next: Ref<*rt-entry*>, LRU-previous: Ref<*rt-entry*>]

A point cache is accessed via the following interface:

Method name	**Parameter types**	**Return type**
new:	*integer*	→ void
delete:	void	→ void
insert:	Ref<*point_impl*> × *rt-entry*	→ void
evict:	*point_impl*	→ void
retrieve:	*point_impl* × *instant*	→ Ref<*point3D*>

An application can create a point cache by calling the constructor with a size argument that specifies the capacity of the cache (number of temporal points). Method "insert" populates the cache with key/value-pairs. Method "evict" removes a key/value-pair; it has no effect if called with an argument not present in the cache. Method "retrieve" returns the requested point coordinates, if resident in main memory and accessible from the cache; otherwise it returns null.

5.6.4 Simplex Managers

The task of a simplex manager is to store the headers for the simplex types *segment*, *triangle*, and *tetrahedron* objects. There is one type of simplex manager for each of these simplex classes, yielding the classes *segmentCluster*, *triangleCluster*, and *tetrahedronCluster*. Applications can create an arbitrary number of simplex managers, facilitating data partitioning. The layout of simplex headers has been described in section 5.6.1. The interface to access a simplex manager is as follows:

Method name	**Parameter types**	**Return type**
new:	void	→ void
insert:	*temporalElement* × Ref<*point*>[2/3/4]	→ void
remove:	*simplex_impl*	→ void

5.7 Storing Temporal Meshes

A temporal mesh data structure implements the interfaces of the types *polyline*, *triangleNet*, and *tetraNet*. Although as many data structures are to be realised, the subsequent presentation is independent of dimension and refers to a "temporal mesh". The mapping onto the specific data structures is then straightforward.

Thereby, the interface to be implemented is the user‘s perspective on a temporal mesh, which consists of a temporal element of validity for the lifetime of the temporal mesh object, a set of time-stamped references to simplices, and the operations defined on it (compare also section 4.5.5). At the same time, auxiliary data structures for a mesh implementation are introduced, as is explained below. On that account, not all of these components must be stored explicitly, but they can be derived from the auxiliary structures. In particular, the temporal element for the lifetime of a mesh object is not stored explicitly. The time-stamped references to simplex objects are stored as *mesh_simplex* objects, the structure of which is defined below. Again, the time-stamp of such a *mesh_simplex* object is derived from internal structures.

Importantly, a temporal mesh corresponds to a certain graph structure. This graph structure, termed temporal graph, is defined and it is shown how a mesh maps onto this graph structure. Many of the problems faced can be solved by algorithms on temporal graphs. For these reasons, it is possible to not having to store time-step extents (sets with simplices which are part of the mesh at each time-step), but to derive the extents from the graph structure. Similarly, maintaining connected components for temporal meshes is beneficial and needs therefore an efficient solution. First, it is beneficial from an application‘s point of view to gain efficient access to the connected components. Second, maintaining connected components enables—in conjunction with the adjacency lists—the traversal of the complete mesh at a given instant.

The temporal graph based approach offers a compromise between compact storage and efficient retrieval. For example, the time-step query demands the temporal mesh at a given instant. This query type has two apparent solutions. First, one could store one list of simplices that are part of the mesh and iterate through this list and output only those simplices that are valid at the given instant. Alternatively, one could store time-step extents that give efficient access to the valid simplices for each time-step. Both approaches have drawbacks. The former suffers from computational costs, since every simplex must be examined. The latter suffers from storage space, since it is expected that many simplices remain from one time-step to the next such that there is redundant storage. This overhead can in parts be alleviated by a versioning approach, but demands a higher computational cost (Shumilov and Siebeck, 2001). Therefore, this thesis proposes to maintain both adjacency lists and connected components. These auxiliary structures facilitate a compromise between the two “extreme” approaches.

The remainder of the section is organised as follows. First, the graph structure, which a temporal mesh corresponds to, is defined. Second, the internal data structure for the graph is explained. Third and based on this structure, it is explained how the problem of maintaining connected components of a temporal mesh can be solved efficiently.

5.7.1 Graph Structure

An atemporal, d-dimensional mesh of d-simplices can be mapped onto a graph structure (compare also section 3.2.3). The d-simplices being part of the mesh correspond to nodes, whereas the neighbourhood-relationships among simplices correspond to arcs. The neighbourhood-relationship between d-simplices holds, if the simplices have a $d-1$-simplex in common.

The graph structure can be extended for the case of temporal meshes. By time-stamping

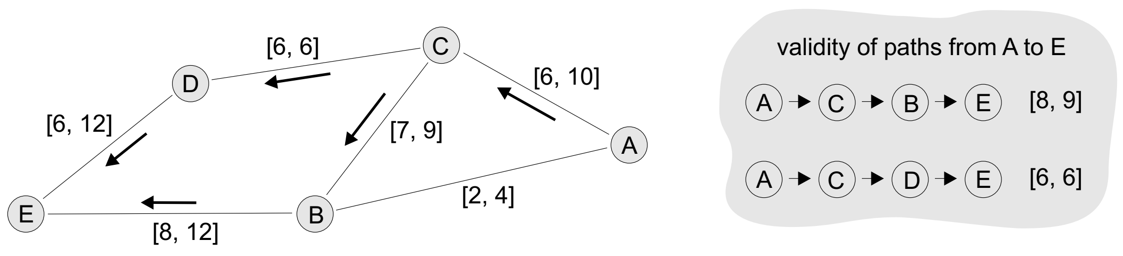


Figure 5.9: Example of a temporal graph and two paths through it from node A to E.

nodes and arcs, the graph structure becomes temporal.

5.7.1. Definition. (*temporal graph*) An undirected *temporal graph* G is a pair $G = (N, A)$ consisting of a set of time-stamped nodes N and a set of time-stamped arcs

$$A \subset \{(\{n_1, n_2\}, e) \mid n_1 \neq n_2 \in N \wedge e \in \mathbb{E}\}, \text{ such that:}$$

$$\text{for all } (\{n_1, n_2\}, e) \in A \quad : \quad e \subseteq \text{lifetime}(n_1) \wedge e \subseteq \text{lifetime}(n_2) \tag{5.1}$$

$$\text{for all } (\{n_1, n_2\}, e), (\{n_1', n_2'\}, e') \in A \quad : \quad \{n_1, n_2\} = \{n_1', n_2'\} \Rightarrow e = e' \tag{5.2}$$

□

For brevity, the braces in the description of an arc will be omitted, such that an arc will be written as "(n_1, n_2, e)". Requirement 5.1 states that an arc may not connect two nodes at times when one of its end-nodes is not valid. Requirement 5.2 states that for each pair of nodes there is at most one connecting arc in A.

An "ordinary" graph can be defined as a projection onto a given temporal interval. This graph contains those nodes and edges the lifetime of which intersects a given interval.

5.7.2. Definition. (*Projection of a temporal graph*) Let $G = (N, A)$ be a temporal graph and $i \in \mathbb{I}$ a temporal interval. The projection $G(i)$ of G onto the interval i is an atemporal graph $G(i) = (N(i), A(i))$ defined as

$$\begin{aligned} N(i) &= \{n \in N \mid \text{lifetime}(n) \cap i \neq \emptyset\} \\ A(i) &= \{\{n_1, n_2\} \mid (n_1, n_2, e) \in A \ \wedge \ i \cap e \neq \emptyset\} \end{aligned}$$

The shorthand $G(t)$ will be used to denote the graph $G([t, t])$ for an instant t. □

5.7.3. Definition. (*path through G, validity of a path*) Let $G = (N, A)$ be a temporal graph. A sequence $p = (n_1, \ldots, n_k)$, $k > 1$, of nodes in N is called a *path through G*, if

$$\text{for all } l = 1, \ldots, k-1 : e \text{ exists such that } (n_l, n_{l+1}, e) \in A \text{ and} \tag{5.3}$$

$$\bigcap_{l=1}^{k-1} \{e_l \mid (n_l, n_{l+1}, e_l) \in A\} \neq \emptyset \tag{5.4}$$

The path is said to be *valid* for each instant in validity$(p) := \bigcap_{l=1}^{k-1} e_l$. □

5.7.4. Definition. (*reachability*) Let $G = (N, A)$ be a temporal graph, n, n' two distinct nodes in N, and P the set of all paths from n to n'. The nodes n and n' are said to be *reachable* from each other for all instant in $\text{reachable}(n, n') := \bigcup_{p \in P} \text{validity}(p)$. □

Requirements 5.3 and 5.4 state that there must exist arcs connecting two consecutive nodes on the path and that the intersection of the lifetimes of these arcs may not be empty. Then, there exist instants t such that there is a corresponding "ordinary" path through $G(t)$. An example for a temporal graph and paths through it is depicted in figure 5.9.

The mapping from a temporal mesh $\mathcal{C}[S, e]$ onto a temporal graph $G = (N, A)$ can be defined as follows[8]:

$$N = S \text{ and } A = \{(s_1, s_2, e) \mid \forall t \in e : s_1(t) \leftrightarrow s_2(t) \;\wedge\; \forall t \notin e : \neg s_1(t) \leftrightarrow s_2(t)\}$$

For the reasons given above, this graph structure is materialised internally for each temporal mesh. For this purpose, several data structures for graphs are available, for instance incidence lists, adjacency lists, or adjacency matrices (see for example Tarjan, 1983; Cormen et al., 1989; Jungnickel, 1990). Here, the adjacency list representation is chosen, since during computations the neighbours of a node (or simplex) must be accessed fast ruling out incidence lists, while at the same time the fan-out of each node is low ruling out adjacency matrices.

However, the notion of an adjacency list must be slightly modified. Since the fan-out for every t in the lifetime of a node is at most $d+1$ for a temporal d-dimensional mesh (a d-simplex has $d+1$ faces and therefore at most $d+1$ neighbours, compare definition 3.2.11), every node is accompanied with a $d+1$-tuple. Since in a temporal mesh the neighbours of a simplex are subject to change, each component of the $d+1$-tuple is a table that allows for retrieving the neighbour of the simplex for the respective face at a given instant. Such a table of simplex s for face f, termed *adjacency table* of s for f, contains triples [*instant*, *bool*, *mesh_simplex*] (see figure 5.10). The table is ordered by the *instant* of each triple. The instants mark off changes in the neighbour-relationship. The *bool* specifies whether the neighbour-relationship given in the triple holds *including* the given instant (true) or not (false). Furthermore, in addition to the nodes given through the above definition of the graph, two special nodes are introduced. First, there is the node that represents the spatial exterior of the mesh, denoted by ◎. Then, an entry in the adjacency table can refer to ◎ as a neighbour, which in fact means no neighbour for the respective interval. Second, there is the node that represents non-existence of the simplex, denoted by ⊠. Then, an entry in the adjacency table can refer to ⊠ as a neighbour, which in fact means that the simplex is not part of the mesh for the respective interval. Recall that the membership of a simplex within a mesh is time-stamped and that these time-stamps are not stored explicitly. The purpose of node ⊠ is the ability to reconstruct this time-stamp.

On the whole, every d-simplex of a d-dimensional temporal mesh has associated with it $d+1$ adjacency tables, one for each face. The table for face f of simplex s can be accessed by the method call

$$s.\text{Adj}(f)$$

[8]Recall that $\leftrightarrow$ denotes the neighbour-relationship between simplices

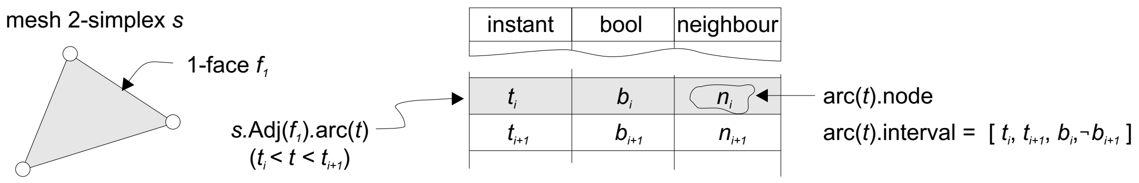


Figure 5.10: A d-simplex that is part of a temporal mesh has associated with every $d-1$-face an adjacency table as depicted on the right hand side. It is shown how to access the relevant information about the neighbourhood-relationship for an instant t.

Each table contains arcs represented by simplex/interval-pairs. By definition, there is exactly one arc for each face f at every single instant in the lifetime of a simplex s. This arc, at a given instant t, can be obtained by the method call

$$\text{Adj}(f).\text{arc}(t)$$

The target node and the interval of an arc can be obtained as follows:

$$\begin{gathered}\text{arc.node}\\ \text{arc.interval}\end{gathered}$$

A new arc to a neighbouring simplex s' for an interval i can be assigned by

$$s.\text{Adj}(f).\text{assign}(s',\ i)$$

This operation assigns mesh simplex s for all instants in i and it updates the interval i' of any other entry by $i' \setminus i$.

The remainder of the section presents a modified version of the general-purpose procedure *breadth-first search* (BFS). Adapted to the temporal graph structure defined above, the procedure is subsequently used to perform certain tasks on each visited node/simplex. In short, the BFS explores those nodes of a graph that are reachable from a given starting node via the arcs incident to any visited node. The order of nodes during the visit is governed by a queue: when visiting a node, all non-visited adjacent nodes are pushed into the queue, whereupon the next node is popped off the top of the queue and visited next.

In contrast, the algorithm for a temporal graph differs from the original BFS in two important ways. First, an additional argument to the algorithm specifies the temporal interval i for the visit: a node is visited if and only if it is reachable (definition 5.7.1) from the starting node during i. Second, a node can be visited more than once, since the time-stamp of an incoming arc must be taken into account when pushing a node onto the BFS-queue; therefore, the usual BFS-flags are turned into temporal elements that specify when a node has been visited, and only those outgoing arcs are followed that intersect with the lifetime of the incoming arc.

Algorithm 5.7.1 BFS(*simplex start*, *interval timeWindow*)

```
 1: $Queue<[simplex, temporalElement]>\ queue$
 2: $queue$.push([$start$, $timeWindow$])
 3: mark($start$, $timeWindow$)
 4: while $queue \neq \emptyset$ do
 5:    $current \Leftarrow queue$.pop()
 6:    process $current$
 7:    for all $f \in current$->simplex->faces do
 8:       for all $arc \in current$->simplex.Adj($f$) do
 9:          if $arc$.interval $\cap$ $current$->temporalElement $\neq \emptyset$ then
10:             when_visit $\Leftarrow$ ($arc$.interval $\cap$ $current$->temporalElement) $\setminus$ get_mark($arc$.node)
11:             if when_visit $\neq \emptyset$ then
12:                $queue$.push([$arc$.node, when_visit])
13:                mark($arc$.node, when_visit)
14:             end if
15:          end if
16:       end for
17:    end for
18: end while
```

The procedure for the "temporal" breadth-first search is depicted in algorithm 5.7.1. The procedure makes use of two transient data structures. First, it uses a queue for the nodes to be visited. Second, it uses a hash table that maps a node to a temporal element. The temporal element indicates for which instants a node has been visited. The hash table is accessed via the functions "get_mark" and "mark" (lines 3, 10, and 13). As mentioned above, the BFS-queue holds simplex/temporalElement-pairs (line 1) and is initialised with the starting simplex and the interval given as a parameter to the procedure (line 2). The while-loop starting in line 4 processes the elements of the BFS-queue as follows. The leading element is popped off the queue (line 5) and can be processed (line 6). Then, the neighbouring simplices are explored, ranging over each face (for-loop starting in line 7) and for each face exploring the adjacency table for neighbouring simplices (for-loop in line 8). Line 9 ensures that only those neighbours are examined that are relevant for the current visit. Line 10 obtains the time for the visit of the neighbour. If there are instants left that are relevant (line 11), the simplex is pushed onto the queue along with this "relevance"-interval (line 12) and is marked accordingly (line 13).

This "general purpose" procedure is used below and serves the purpose to explore a connected component of a temporal mesh, as defined in the next section.

In summary, this section introduced the notion of a temporal graph. It was shown to be an adequate abstraction for the (discrete) structure of a temporal mesh by specifying a mapping from a temporal mesh onto this graph structure. Furthermore, the section introduced the necessary data structures to represent the temporal mesh as a graph. Finally, a generalisation of the "general purpose" procedure breadth-first-search to temporal graphs has been presented that is used subsequently to explore a connected component of a temporal mesh. The next section focuses upon this aspect of the connectivity of a temporal mesh and how to manage it efficiently when faced with updates.

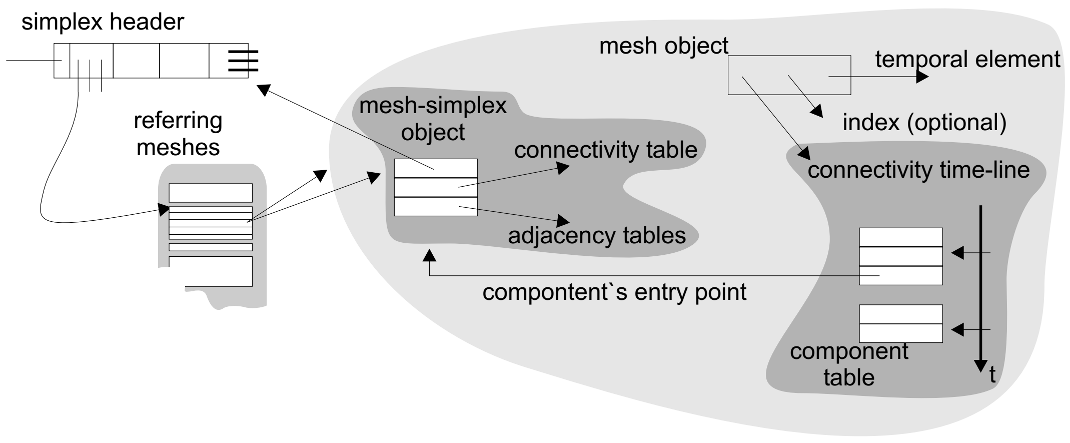


Figure 5.11: The relationship between simplex-headers, mesh_simplices, mesh-objects, and connectivity time-line.

5.7.2 Connected Components

The adjacency lists can be used to traverse all simplices of a mesh for a given instant, for example via breadth- or depth-first-search; however, simplices in different connected components cannot be reached in this way. For "usual" (atemporal) graphs, the set of connected components of an undirected graph is defined as the set of equivalence classes of the "reachable"-relationship between nodes: a node n_1 is reachable from another node n_2, if there exists a path along arcs from n_2 to n_1 (Tarjan, 1983; Jungnickel, 1990). Likewise, in a temporal graph the connected components are defined in the same way for each atemporal "instance"[9]. During the lifetime of a temporal graph, several events are of interest: connected components can appear, disappear, merge, and split. A connected component in a temporal graph $G = (N, A)$ is a pair (C, i), where $C \subseteq N$ is a set of nodes and i is a temporal interval of validity such that for each instant $t \in i$ the set of valid nodes in C

$$\{n \mid n \in C \text{ and } t \in n.\text{lifetime}\}$$

is a connected component in $G(t)$. The boundaries of the interval i correspond to the mentioned events that are defined subsequently.

In a non-dynamic, atemporal environment, where "insert"- and "remove"-operations are absent, it would suffice to once compute the set of connected components for a given mesh and to store a representative simplex for each. Here, however, the problem is more complex. The goal is to store the set of connected components for every instant the connectivity changes at. These instants are maintained in a separate data structure, called *connectivity time-line*. Then, the mesh at instant i can be computed by retrieving the connected components of the last instant before i within the connectivity time-line and traversing each component using the adjacency lists of the nodes. The structure of a mesh object is depicted in figure 5.11.

[9]Although it is possible to extend the definition of a connected component to a temporal graph G, it is more important to focus upon connected components for the atemporal graphs $G(t)$.

Furthermore, the problem of maintaining the data structure on updates to the mesh must be solved. For a simpler situation the problem is an instance of the well-known *union-find*-problem (or disjoint-set-problem; Tarjan, 1983; Cormen et al., 1989): the atemporal case with only the “insert”-operation supported. Originally, the union-find-problem demands the maintenance of a partition of a set of objects and the support for the following operations: (1) “union(x, y)”, combining the partitions that the objects x and y belong to; (2) “find(x)”, retrieving the partition x belongs to. In parlance of an atemporal mesh, the partitions correspond to the connected components, objects to simplices, and operation “union” to the insertion of a simplex that connects previously disconnected components.

The asymptotically fastest union-find data structure known is the disjoint set forest with the heuristics “union by rank” and “path compression” (Tarjan, 1983; Cormen et al., 1989). In this data structure, each disjoint set is given by a representative that belongs to the set. Each object is element of exactly one tree in the disjoint set forest and carries a reference to its parent in the tree. The root of the tree is the representative object for the disjoint set. Hence, the “find(x)”-operation is performed by following the parent-references up to the root of the tree that x belongs to. The “union(x,y)”-operation is performed simply by letting the root of x point to the root of y or vice versa. Roughly, the “union by rank”-heuristic states that during “union” the tree with the fewer nodes is chosen to point to the root of the other tree. The “path compression”-heuristic states that during “find“ all visited nodes are adapted to point to the root directly, accelerating the next “find”-operation for these nodes.

For the management of connected components of a temporal mesh, the union-find data structure and its algorithms are extended by time. The new structure can be summarised as follows. The temporal aspect must be reflected such that for every instant in a mesh‘s lifetime the set of connected components is available. Therefore, the modified structure stores a list of those timesteps the connectivity changes at. This list is called the *connectivity time-line* and is in particular defined to contain an instant t, if and only if

- a component appears or disappears at t, or
- components split or merge at t

The following definitions specify these events precisely. Expressed in terms of temporal graphs, they can be mapped to temporal meshes straightforwardly. The definitions make use of the notation e° for a temporal element e, which denotes the interior of e, i.e., all instants in e except its boundary instants.

5.7.5. Definition. (*appearance/disappearance of a component*) Let $G = (N, A)$ be a temporal graph. A component C is said to *appear at* instant t in G, if and only if the following conditions hold:

1. There is a node $n \in C$ and a left-closed interval $i \in n$.lifetime such that $i^- = t$.
2. For all nodes $n' \in N$ with $t \in n'.\text{lifetime}^\circ$ there is no path from n to n' in $G(t)$.

Similarly, a component C is said to *appear after* t in G, if and only if the following conditions hold:

1. There is a node $n \in C$ and a left-open interval $i \in n$.lifetime such that $i^- = t$.
2. For all nodes $n' \in N$ with $t \in n'$.lifetime$^\circ$ there is, for $\epsilon \to 0$, $\epsilon > 0$, no path from n to n' in $G(t + \epsilon)$.

The disappearance of a component is the reverse of appearance of a component. □

The first condition in "appearance at" and "appearance after" in the definition above expresses that a node starts its lifetime with a closed interval and with an open interval, respectively. The second condition expresses that a node n, starting its lifetime at t, is not reachable from any node that started its lifetime before t. Of course, n might be reachable from such a node at an instant $t' > t$, caused by a merge event.

5.7.6. Definition. (*split/merge of components*) Let $G = (N, A)$ be a temporal graph. A component C is said to *split* in G at instant t, if and only if the following condition holds:

- There are nodes $n \neq n' \in C$ such that, for $\epsilon \to 0$, n and n' are reachable at $t - \epsilon$, but not at t.

Similarly, component C is said to split after t in G, if and only if the following condition holds:

- There are nodes $n \neq n' \in C$ such that, for $\epsilon \to 0$, n and n' are reachable at t, but not at $t + \epsilon$.

A merge of components is the reverse of a split of components. □

Importantly, the event of an "extension" of a connected component (when a node starts its lifetime, but does not connect previously unconnected components) is not among the events defined above. The same holds for a reduction of a connected component. Such events need not be recorded by the connectivity time-line. An example for some of these events is depicted in figure 5.12.

The events (dis-)appearance, split, and merge, determine the connectivity time-line, which is maintained by the union-find operations (triggered by "*mesh*::insert(*simplex*, *temporalElement*)", see below), as well as by the operation "*mesh*::remove(*simplex*)" that can cause a split of components. The connectivity time-line is a data structure that allows to associate each instant with a set of *entry-points* into the mesh. The entry-points correspond to the component representatives. The data structure supports the operations

- insert(*mesh_simplex s*, *interval i*): associates each instant in i with s. If s was associated with the instants in the temporal element e, then s is associated with the instants in $e \cup i$ after calling this function.
- remove(*mesh_simplex s*, *interval i*): removes the association. If s was associated with the instants in the temporal element e, then s is associated with the instants in $e \setminus i$ after calling this function.

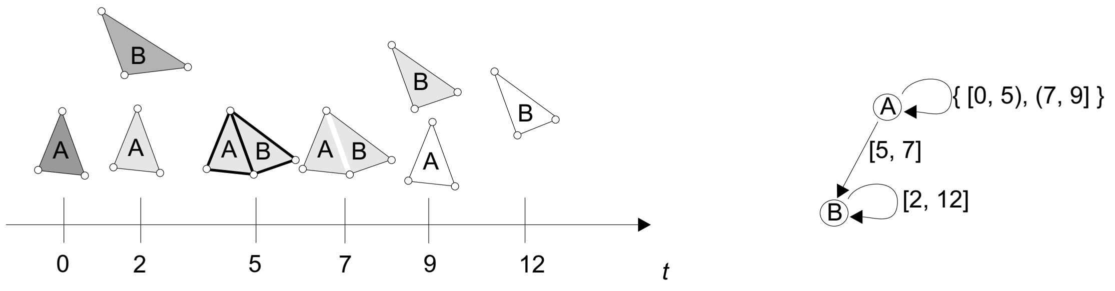


Figure 5.12: After inserting simplex B into the mesh, several events had to be recorded on the time-line (left part). First, a new component comes to existence at instant 2. At instant 5, A and B become neighbours and their components must be merged. At instant 7, the neighbour-relationship ends. At instant 9 and 12, the components of simplex A and B leave the mesh, respectively. The right part shows the connectivity graph (or disjoint set forest), in which the arcs point to the parent of a component.

Furthermore, the disjoint set forest of the union-find data structure is altered such that each node points to the "correct" parent *for every instant* in the node's lifetime. In analogy to an adjacency table (see also figure 5.10, page 90), each simplex s is accompanied by a *connectivity table* that can be accessed by

$$s.\text{Conn}$$

This table is structured like an adjacency table; the special node ◎, however, is not needed in a connectivity table. The update operation ("assign") is the same as for an adjacency table. The parent simplex for an instant t can be obtained by

$$s.\text{Conn->parent}(t)$$

In summary, the modified union-find data structure supports the following operations. The

Algorithm 5.7.2 *mesh*::make_component(*mesh_simplex s*) → *void*

```
1: for all intervals $i \in s$.lifetime do
2:     this->connectivityTimeline->insert($s$, $i$)
3:     $s$->Conn->assign($s$, $i$)
4: end for
```

operation "make_component" (algorithm 5.7.2) performs the following tasks. First, it updates the connectivity time-line of the mesh (line 2). Second, it initialises the connectivity table of the argument mesh_simplex s (line 3). The connectivity table of s is modified such that for each interval in s.lifetime the arc to the parent of s is directed to s itself (s is its own representative).

The operation "make_link" (algorithm 5.7.3), invoked by "union", updates the connectivity tables of s_1 and s_2 by making s_1 the parent of s_2 for the interval i.

Algorithm 5.7.3 *mesh*::make_link(*mesh_simplex* s_1, *mesh_simplex* s_2, *interval* i) $\rightarrow$ *void*

```
1: $s_1$.Conn->assign($s_2$, $i$)
```

Algorithm 5.7.4 *mesh*::find(*mesh_simplex* s, *interval* i) $\rightarrow$ List<[*mesh_simplex*, *interval*]>

```
1: List<[$mesh\_simplex$, $interval$]> $C \Leftarrow \emptyset$
2: for all $arc \in s$.Conn where $arc$.interval $\cap i \neq \emptyset$ do
3:    if $arc$.node=$s$ then
4:       insert [$arc$.node, $arc$.interval $\cap$ $i$] into $C$
5:    else
6:       List<[$mesh\_simplex$, $interval$]> $comps \Leftarrow$ find($arc$.node, $arc$.interval)
7:       for all $c \in comps$ do
8:          $s$.Conn->assign($c$.mesh_simplex, $c$.interval)
9:       end for
10:      insert each $c \in comps$ into $C$
11:   end if
12: end for
13: return $C$
```

The operation “find” (algorithm 5.7.4) retrieves the list of component representatives of s that are valid during the given interval i. Each representative is accompanied with the information, when it was the representative for the component (limited to the interval i). The procedure implements the path-compression heuristic for the temporal case (for-loop in line 7). The strategy is a depth-first-search through the disjoint set forest, using the connectivity tables of the simplices. In line 6, the algorithm contains a recursion that terminates at component representatives (“if” in line 3). The operation is a *two-pass* operation that in its first pass finds a representative simplex of the component (line 6) and in its second pass updates the arcs of any visited simplex to point directly to the representative (line 8).

Algorithm 5.7.5 *mesh*::union(*mesh_simplex* s_1, *mesh_simplex* s_2, *interval* i) $\rightarrow$ *void*

```
1: List<[$mesh\_simplex$, $interval$]> $C_1 \Leftarrow$ $this$->find($s_1$, $i$)
2: List<[$mesh\_simplex$, $interval$]> $C_2 \Leftarrow$ $this$->find($s_2$, $i$)
3: for all pairs $(c_1, c_2) \in C_1 \times C_2$ with $c_1$.interval $\cap$ $c_2$.interval $\neq \emptyset$ do
4:    $interval$ $j \Leftarrow c_1$.interval $\cap$ $c_2$.interval
5:    $this$->connectivityTimeline->remove($c_2$.mesh_simplex, $j$)
6:    make_link($c_1$.mesh_simplex, $c_2$.mesh_simplex, $j$)
7: end for
```

The operation “union” (algorithm 5.7.5) connects the components of s_1 and s_2 for every instant in i. It first computes the list of all component representatives for s_1 and s_2, valid during the interval i by calling “find” (lines 1 and 2). Each component representative is then associated with its interval of validity, indicating when it is the representative for the component. Method “union” must consider all pairs (c_1, c_2), $c_1 \in C_1$, $c_2 \in C_2$, with intersecting intervals (line 3) and make c_1 the representative of c_2 during the intersection interval (line 6). The component representative c_2 is removed from the mesh‘s connectivity time-line for the intersection interval (line 5).

The operations “make_component”, “union”, and “find” are used subsequently to perform update operations on temporal meshes that are focus of the following section.

5.8 Update-Operations

Update operations are intended for creating, deleting, and modifying spatio-temporal objects. Modifications are, for instance, insertion/removal of simplices from temporal meshes or the modification of a point‘s trajectory. Update operations must ensure the geometric constraints mentioned in section 3.2.3. Furthermore, updates of temporal simplices can trigger updates of the internal structures of temporal meshes. In particular, adjacency tables can be affected, since the neighbour-relationship between the simplices of a mesh can change. For the same reason, the connectivity of a mesh can change, necessitating an update of its connectivity time-line and connectivity tables of its simplices. Hence, these operations are computationally more demanding than usual collection updates.

Updates of spatio-temporal objects can change the geometry-part (location of vertices), the temporal part (specification of lifetime), or both. Temporal constraints are violated if after an update to the temporal part of

1. a temporal point p: there is a d-simplex $s = [\{p_1, \ldots, p_{d+1}\}, e]$ such that $p \in \{p_1, \ldots, p_{d+1}\}$ and $e \supset p.\text{lifetime}$
2. a temporal d-simplex $s = [\{p_1, \ldots, p_{d+1}\}, e]$:
   (a) there is a point in $\{p_1, \ldots, p_{d+1}\}$ violating condition 1.
   (b) there is a mesh m containing a time-stamped simplex-reference $[s, e]$ such that $e \supset s.\text{lifetime}$

Constraints for the geometry-part of a spatio-temporal object can only be violated by inserting a simplex into a mesh and by updating temporal points. Precisely, geometry constraints are violated after an update if (1) there is a simplex violating the restrictions of definition 3.2.9 or if (2) there is a mesh violating the restrictions of definition 3.2.11.

5.8.1 Creation and Deletion

Creation and deletion of the temporal simplex objects have been described above. Geometry-constraints for temporal simplex objects can be checked with the operations described in section 6.2.3. The creation of a mesh object is straightforward, while the deletion is not. Indeed, referential integrity must be maintained and the internal structures of a mesh cannot be deleted without further ado. For each simplex in the mesh to be deleted, the mesh must be removed from the set of mesh references (in the object header for the simplex, see section 5.6.1). Therefore, the strategy for the deletion of a mesh is to first perform a BFS through the mesh (cf. previous section) and to update the simplex headers accordingly.

5.8.2 Updating Temporal Simplices

An update of the geometry-part of a temporal point is processed as follows. Before a point manager updates its internal structures, the geometry constraints for all referring objects are checked to determine if the update can be permitted. The objects that are affected by the update are given by the materialised reference-graph. Hence, all simplices, which hold a reference to this point, are stored along with the updated point object and must be checked. Along with each such simplex, the set of referring temporal meshes is stored and must be checked. If any of these objects responds with a failed check, the update is refused and an exception is raised. If all checks are passed, the update can be processed. Using again the materialised referring graph, the update is propagated to the referring objects, in order to maintain indices owned by any referring mesh.

Similarly, an update of the temporal part of any of the simplex objects can be checked, using the conditions given above. The remainder of the section focuses on the update of temporal meshes.

5.8.3 Inserting a Simplex into a Mesh

Given a temporal mesh object, the insert operation can be used to insert a time-stamped reference to a simplex into the mesh. An insert can affect the connectivity of the mesh: it can create a new component, merge previously unconnected components, or leave the connectivity unchanged (by only enlarging a component). These events can occur several times for each inserted simplex. Therefore, the insert-operation must compute these events and record the changes into the connectivity time-line and connectivity tables. Since geometric constraints can be violated by this operation (see definition 3.2.11 of a temporal mesh), the operation must keep a check on this circumstance. On fail, the operation exits by raising an exception. On success, internal structures must be updated, in particular the connectivity time-line, the connectivity tables and the adjacency tables.

The necessary steps are depicted in algorithm 5.8.1. For presentational reasons it is assumed that the simplex to be inserted is not already contained in the mesh; recall from chapter 4 that "insert" allows to change the containment temporal element and the algorithm must therefore be revised slightly. However, with this assumption, the steps to be performed include: (1) check if geometric constraints for the particular datatype are violated by the update (line 1); (2) if not, update adjacency tables of the mesh (lines 5–11); (3) insert the simplex into the index of the mesh, if present (lines 12–14); (4) handle connected components (lines 15–20) by creating a new component for s and then calling for each neighbour in each face the union operation described in section 5.7.

5.8.4 Removing a Simplex from a Mesh

Given a temporal mesh object, the remove operation can be used to remove a simplex from the mesh. While geometric constraints cannot be violated by this operation, internal structures must be updated, in particular the connectivity time-line, the connectivity tables, and the adjacency tables.

Algorithm 5.8.1 *mesh*::insert(*simplex s*, *temporalElement e*)

```
1: if $[s, e]$ violates mesh restrictions then
2:     throw exception
3: end if
4: $ms \Leftarrow$ new $mesh\_simplex(s)$
5: for all $f \in ms$->$faces$ do
6:     Set<[mesh_simplex, interval]> neighbours $\Leftarrow \emptyset$
7:     neighbours $\Leftarrow$ this->get_neighbours($ms$, $e$)
8:     for all $n \in$ neighbours do
9:         $ms$->Adj($f$)->assign($n$.mesh_simplex, $n$.interval)
10:    end for
11: end for
12: if this->has_index then
13:     this->rTree->insert($ms, e$);
14: end if
15: this->make_component($ms$, $e$)
16: for all $f \in ms$->faces do
17:     for all $arc \in ms$->Adj($f$) do
18:         this->union($ms$, $arc$.simplex, $arc$.interval)
19:     end for
20: end for
21: $s$->internal_part->insert_referent($ms$)
```

The removal of a time-dependent simplex s from a mesh m is accomplished as follows. It is assumed that s is contained in m. (This can be checked by examining s‘s object header; the operation raises an exception if s is not contained in m.) The strategy for the operation is then as follows.

1. Update the adjacency tables for all neighbours of s
2. Compute when the removal of s causes a change in connectivity for the connected components which s resides in; result is a temporal element e
3. Recompute the connected components of each neighbour of s for the instants in e and update the connectivity time-line and connectivity tables

The details of these steps are depicted in algorithm 5.8.2. In step 1 the adjacency tables of each neighbour n of s are updated by setting the reference within n to null where it references s (summarised in line 1). It is realised by iterating through each entry of the adjacency tables for s. Let s' denote a neighbour of s, i the interval of the neighbourhood-relationship and f' the face of s' that is common with s (neighbour-relationship). Then the operation calls s'->Adj(f').set($\circledcirc, i$). Step 2 is a non-trivial operation and makes use of procedure “when_reachable” (line 6, described below). This step iterates over all faces of s (line 2) and for each such face over all arcs of the face‘s adjacency table (line 3). It follows from the neighbourhood-relationship that for each instant in the validity of the arc, s and its neighbour reside in the same component. Line 4 retrieves the representatives for these components. For each component found (line 5), the procedure "when_reachable" computes when the neighbour is still connected with the component representative (after removing s from the neighbourhood-graph), resulting in a temporal element e. Then the complement $\overline{e}$ of e, intersected with the component's lifetime, contains exactly those instants for which the component is split. Put differently, before the removal of s each path valid during $\overline{e} \cap c$.lifetime from the neighbour of s to the component representative contains s. Step 3 comprises the for-loop in line 7 that iterates over all intervals during which the "original" component is split into several new ones. The loop contains the necessary actions of updating the connectivity time-line (lines 8 and 9) and the connectivity tables (line 10). In particular, the latter action is performed by procedure BFS. The BFS is started at the neighbour of s, which is made the representative of the new component. The connectivity table of each visited simplex is updated to reflect that change (parent pointers are adjusted to point directly to the new representative simplex).

Algorithm 5.8.2 *mesh*::remove(*simplex s*)

```
 1: update neighbours of $s$     // details see text
 2: for all $f \in s$->faces do
 3:    for all $arc \in s$->Adj($f$) do
 4:       List<[simplex, interval]> components ⇐ this->find($s$, arc.interval)
 5:       for all $c \in$ components do
 6:          temporalElement $e$ ⇐ when_reachable($c$.simplex, arc.node)
 7:          for all intervals $j \in \overline{e} \cap c$.interval do
 8:             this->connectivityTimeLine-> remove($c$.simplex, $j$)
 9:             this->make_component(arc.node, $j$)
10:             BFS(arc.node, $j$, call $s'$->Conn.assign(arc.node, $j$) on each visited simplex $s'$)
11:          end for
12:       end for
13:    end for
14:    $s$->remove_referent(this)
15: end for
```

The algorithm for "remove" uses sub-procedure "when_reachable" to compute at what instants a path exists between two simplices. The pseudocode for this procedure is shown in algorithms 5.8.3 and 5.8.4. In analogy to the algorithm BFS, this procedure uses the adjacency tables to perform its task; however, the overall strategy is a depth-first search through the temporal mesh. Algorithm 5.8.3 starts the recursive algorithm 5.8.4 for each neighbour of simplex s_1. These neighbours are taken from the adjacency tables of s_1 (for-loops in lines 3 and 4). Algorithm "recursive_reachable" uses a transient hash table that maps simplices to the temporal element for which the simplex has already been visited (DFS-flag). The table is accessed by the functions "get_mark" and "mark". For presentational reasons, the table is a global variable that can be accessed in recursive calls to "recursive_reachable"[10].

Using a recursive design, the strategy is depth-first. Each call of "recursive_reachable" (line 5) aims at finding a different path from s_1 to s_2 and it returns with the validity of the path found (or with an empty *temporalElement* if no more path is found). After such a call,

[10]A better solution is to create the table in algorithm "when_reachable" and to pass it as an additional parameter to "recursive_reachable".

Algorithm 5.8.3 when_reachable(*simplex* s_1, s_2) $\rightarrow$ *temporalElement*

```
1: mark($s_1$, $[-\infty, \infty]$)
2: temporalElement result $\Leftarrow \emptyset$
3: for all $f \in s_1$->faces do
4:   for all $arc \in s_1$->Adj($f$) do
5:     $result \Leftarrow result \cup$ recursive_reachable($arc$.simplex, $s_2$, $arc$.interval $\setminus$ $result$)
6:   end for
7: end for
8: return result
```

the next call to "recursive_reachable" can be performed excluding the result found thus far (subtracting *result* from the validity of the neighbour reference, line 5 of algorithm 5.8.3).

Algorithm 5.8.4 ("recursive_reachable") aims at finding all paths from s to *target*, limited to the temporal interval i. It returns with the union of the validity of these paths, limited to i. To this end, it visits its first parameter s. It checks, if s is equal to the target simplex (lines 1-3). If so, it returns with its third parameter i. Otherwise, the procedure explores the relevant neighbours of s (for-loops in lines 6 and 7). Algorithm "recursive_reachable" calls itself recursively on the currently explored neighbour, the target simplex *target*, and the relevance interval (line 10). Again, the result found thus far can be excluded from consideration, since with the length of a path (number of nodes), the validity time of the path cannot grow (see also definition 5.7.3).

Algorithm 5.8.4 recursive_reachable(*simplex* s, *target*, *interval* i) $\rightarrow$ *temporalElement*

```
1: if $s$ ==target then
2:   return $i$
3: end if
4: temporalElement result $\Leftarrow \emptyset$
5: mark($s$, $i$)
6: for all $f \in s$->faces do
7:   for all $arc \in s$->Adj($f$) do
8:     $timeWindow \Leftarrow (arc.\text{interval} \cap i) \setminus$ get_mark($arc$.simplex)
9:     if $timeWindow \neq \emptyset$ then
10:      $result \Leftarrow result \cup$ recursive_reachable($arc$.simplex, $target$, $timeWindow \setminus result$)
11:    end if
12:  end for
13: end for
14: return result
```

5.9 Snapshot Query

Having described the storage structures for spatio-temporal objects, the focus is now on a realisation of a basic retrieval operation: the time-step query, which retrieves a pure-spatial object given an instant in time and hence demands to visit all mesh elements valid at that instant; the snapshot query implements method "at" of the interface of class *stObject* (figure 4.3, page 53).

The description can be brief, as the method uses the above algorithms. If the method is invoked such that also point coordinates are requested for the particular instant, a point-cache instance is created (section 5.6.3). The connectivity time-line of the mesh is searched for the biggest instant less than the query instant. The algorithm loops over all components associated with the instant. For each component, the valid entry point simplex s for the query instant t is retrieved and the BFS routine is invoked with s and $[t, t]$ as parameters. During the BFS procedure, each time the neighbour of a simplex has to be *visited* (pushed into the BFS queue), the neighbour is chosen which has been temporally valid at t. Each time a simplex has to be *examined* (popped from the BFS queue) it is inserted into the resulting mesh object and is marked visited. If requested, point objects are created, valid at instant t.

Chapter 6

Operations for Spatio-Temporal Objects and their Realisation

The conceptual model, the data types, and the data structures presented in the previous chapters will prove useful only if they are accompanied by retrieval facilities. Some of them have also been introduced in these chapters. Among them are the *construction-, destruction-* and *update*-operations. Likewise, support for visualisation was given by *snapshot queries.* More query facilities should be offered in spatio-temporal database systems; however, implementation concepts have been seldom proposed in literature. This chapter develops implementation concepts for the spatio-temporal range query and for intersects- and distance-based predicates. Furthermore, it is shown how to ensure the restrictions on spatio-temporal objects.

The necessary steps for the realization of the operations are being developed: (1) geometric base computations on primitive types (defined below), to be described in a mathematical framework; (2) application of these computations for compound types, like the trajectory of a temporal simplex or a temporal mesh. Moreover, the notion of *spatial filtering* via minimum bounding boxes, widespread in spatial databases, is transferred to the spatio-temporal setting. The operations to be described are generalised to so-called decomposable problems. Furthermore, if present, indexes can be exploited in the computation of operations.

6.1 Conceptual Considerations

In general, computing elementary binary operations between temporal simplices cannot be reduced to a corresponding operation on atemporal simplices, for example, through computation at each temporal snapshot using well-known spatial operations. Hence, spatio-temporal operations are more involved. Operations are to be derived that operate upon certain restrictions on temporal simplices, the so-called spatio-temporal primitive objects (defined below). This section clarifies what makes up $O(1)$ operations on spatio-temporal objects and how to combine them to operations on the spatio-temporal data types. It also shows how to iterate over the primitive objects of a given spatio-temporal object. Furthermore, this section describes how the proven technique of a spatial filter for geometric operations can be transferred to the spatio-temporal setting.

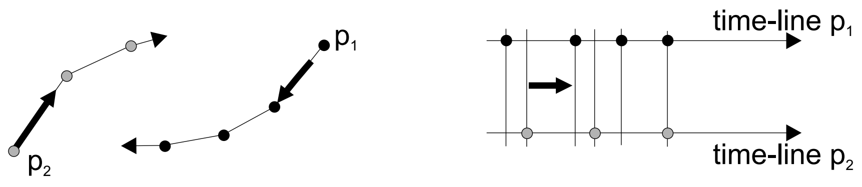


Figure 6.1: The merged time-line of two temporal points induces a sequence of spatio-temporal primitive objects. The bold arrow indicates a time interval for a spatio-temporal primitve object.

6.1.1 A Hierarchy of Operations

Elementary operations form the building blocks for operations on compound spatio-temporal objects, like complete temporal simplex traces or a complete temporal complex. Beforehand, though, it is necessary to go a level below the temporal simplices (Breunig et al., 2002). These operations are designed such that they run in constant time. In the pure-spatial case, these $O(1)$-operations include, for example, point/point-distance, point/segment-distance or segment/segment-intersection, to name but a few. However, the number of time-steps of, for instance, a temporal point is not bounded by a constant. Therefore, an operation accessing a non-constant number of time-steps is not among these atomic operations.

Indeed, computing, for example, the minimum distance between temporal points is not an $O(1)$-operation. To perform such a computation, it is broken up into smaller units, each of which performs an elementary operation (see figure 6.1). Linear moves of a temporal point occur during two consecutive time-steps on its time-line: with linear point movement, only two time-steps are needed to compute the position in between. Then, consecutive time-steps on the *merged* time-line mark time intervals during which *both* temporal points perform linear moves (see bold arrows in figure 6.1). Hence, on this interval temporal simplices can be represented in $O(1)$ space. In general, elementary spatio-temporal $O(1)$-operations apply to temporal simplices, limited to consecutive time-steps on their merged time-lines. Such a limited temporal simplex is called spatio-temporal primitive object:

6.1.1. Definition. (*spatio-temporal primitive object*) Let o denote a spatio-temporal object. The object o is called a *spatio-temporal primitive object*, if

- o is a temporal simplex $s[V, e]$, and
- for each $v \in V$: $v_{|e}$ moves on a straight line, i.e., $\text{gsteps}(v_{|e}) = 2$. □

The rationale behind the definition is as follows. Given two spatio-temporal primitive objects $s_1[V_1, e_1]$ and $s_2[V_2, e_2]$, a primitive operation operates upon these primitives, limited to the interval $e' = e_1 \cap e_2$. A different perspective on primitive operations is through the merged time-line. Given two (complete) temporal simplices s, s', one can merge the time-lines of both objects and choose the intervals for which both objects are defined (see figure 6.1). Importantly, both objects are continuous and differentiable on each such intervals and methods from calculus can be applied to perform basic computations, like the minimum Euclidean distance between the spatio-temporal primitive objects (see subsequent sections). Depending on the particular problem at hand, the results from these basic computations can then be combined to obtain the result for s and s' (operation min in case of the minimum Euclidean distance).

A primitive temporal point can be specified by two 3D points $\mathbf{a} := \mathbf{p}(0)$ and $\mathbf{b} := \mathbf{p}(1)$ to express the temporal point parametrically:

$$\mathbf{p}(t) = \mathbf{a} + t \cdot (\mathbf{b} - \mathbf{a})$$

To sum up, spatio-temporal primitive objects can be obtained when simplices are limited to the interval between two consecutive time-steps on their (merged) time-line. Then, it is possible to reduce the computations of operations to these primitive objects, the movement of which is “well-behaved” and methods from calculus can be applied. The results can be combined to form a result for the given temporal simplices. The results from temporal simplices can be combined to a result for temporal meshes.

6.1.2 Iteration over the gsteps of a Spatio-Temporal Object

Having defined spatio-temporal primitive objects and how they are used in operations, a further building block is the iteration over the gsteps of a spatio-temporal object (section 3.2.3), obtaining primitive objects one by one. The concept to be realised is that of an *iterator*, the *gstepIterator*, offering the methods “open”, “next”, and “close”. Since the *gstepIterator* is to be used only internally, only the internal spatio-temporal classes are allowed to create a respective iterator. The interface of a *gstepIterator* for a d-simplex-implementation is as follows:

Method name	**Parameter types**	**Return type**
new:	void	$\rightarrow$ Ref<*gstepIterator*>
open:	Ref<*interval*>	$\rightarrow$ void
next:	void	$\rightarrow$ Ref<[*point3D*[d+1][2], *interval*]>
close:	void	$\rightarrow$ void

Here, method “next” returns two snapshots of $d + 1$ points in 3D and a temporal interval i, where the first and second snapshot of $d + 1$ points are valid at i^- and i^+, respectively.

The constructor “new” is called by the spatio-temporal object implementations. Method “open” can then be used to open the iterator for a given interval (, i.e., only primitive objects are returned the lifetime of which intersects the interval). Method “next” returns a pair of snapshots, which specify a spatio-temporal primitive object. The method returns null in case all primitive objects have been visited. Method “close” closes the iterator.

The implementation depends upon the particular type of spatio-temporal object. For a *point_impl*-object, the clustering R-tree of its point manager is utilised. Note that for clustering purposes a point trajectory is split into primitive objects. On opening the iterator with an interval i, the R-tree is searched for the first leaf-entry of the point that contains the instant i^-. This search is guided by i^- and the *boundBox4D* of the point header (section 5.6.1).

On calling "next", the current leaf-entry is used to construct the return object and the leaf-entry pointers are used to find the next primitive object for the point.

The implementation for a temporal *segment_impl*, *triangle_impl*, and *tetrahedron_impl* can be described together as d-simplices. A *gstepIterator* for a temporal d-simplex coordinates $d+1$ *gstepIterators* for temporal points (the vertices of the simplex) and stores internally a state variable, which is the instant of the last returned snapshot. A call to "open" creates the $d+1$ iterators for the simplex's points, opens them, calls "next" on each and initialises its state variable with the lower boundary of the interval parameter. On calling "next", two snapshots of the simplex must be created: the first one for the instant given by the state variable and the second one for the smallest upper instant of the points's snapshots. The chosen iterator is advanced by calling "next" and the state variable is set to the chosen instant. This procedure can be seen as a construction of the merged time-line of the $d+1$ vertices.

The implementation for the temporal mesh implementations is described together as d-mesh. A *gstepIterator* for a temporal d-mesh makes use of a transient point cache and aims at exploiting locality-of-reference for the point accesses. Instead of processing the simplices of the mesh one by one, the iterator therefore interleaves primitive objects from different simplices. The iterator outputs primitive objects of a fixed simplex as long as data is in the point cache to create output without a further access to secondary memory. Recall from section 5.6.3 that a point cache pre-fetches point data. If the cache data for a simplex is used up, the iterator chooses a simplex with data available. If none is available, the processing of the next simplex entails an access to secondary storage. The realisation of this strategy is a modified version of the BFS-procedure presented in section 5.7. The differences are as follows. The processing interval of a simplex (as BFS-node) is not bound to the interval of the incoming arc, but to the availability of point-data in the point cache. For I/O-awareness (giving those nodes precedence that have cache data), an additional BFS-queue is introduced. The first queue is used for simplices that have point-data in the cache such that processing these simplices does not necessitate access to secondary storage. The second queue is used for simplices that need further secondary storage access to be processed. Only if the first queue is empty, a simplex is popped off the second queue.

Summing up, the presented iterators are a tool for the implementation of the operations on spatio-temporal objects. Using these iterators on a (complete) spatio-temporal object, its primitive objects can be obtained, which in turn are to be consumed by primitive operations to be presented in the next section. Beforehand, though, the notion of a spatial filter is extended to the spatio-temporal setting.

6.1.3 Extending Spatial Filtering

In spatial databases it is common practice to apply cheap pre-checks to expensive geometric computations. Mostly, the approximation of a *minimum bounding box* (mbb for short) of a spatial object serves this purpose. A 4D minimum bounding box of a spatio-temporal object is defined by four intervals: three spatial intervals, which are the extents of the *trajectory* of the object; one temporal interval, which is the hull of the valid time of the object. Therefore, one can readily apply the technique of spatial filtering for geometric operations to the spatio-temporal objects. With this technique, expensive geometric operations are only performed, if the objects under consideration pass a cheap pre-check (two-step process). For example, if two spatio-temporal objects intersect each other at an instant, then the 4D bounding box approximations of the two objects intersect. The intersection of the 4D boxes is therefore a necessary (but not a sufficient) condition. As such, it can be exploited as a filter by first performing this cheap test and only on success performing the more expensive, exact intersection test.

For primitive objects, though, one can introduce a three-step process for spatio-temporal filtering. The first step is the 4D bounding box test. The second step consists of a further filter operation, but this time based on a more exact approximation. The third step is then the exact geometric computation.

Since in spatial databases the mbb-approach proved successful, the following definition makes sense.

6.1.2. Definition. (*strong mbb criterion*) An approximation scheme for spatio-temporal objects meets the *strong mbb criterion*, if and only if (1) for the approximation of a spatio-temporal object o the projection of this approximation onto an arbitrary instant t is equal to $mbb(o(t))$; and (2) the approximation of a spatio-temporal object can be represented with constant space.

Using 4D bounding boxes, the strong mbb criterion is only met by a very limited set of moving and changing spatial objects.

6.1.3. Proposition. *4D bounding boxes meet the strong mbb criterion for those spatio-temporal objects o, for which there is an mbb b, such that for all instants $t \in time(o)$: $mbb(o(t)) = b$.*

Note that for rigid objects—including moving vertices—the above proposition implies that movement and change is limited, if it can occur at all, if the strong mbb criterion is to be met. Change is possible for non-rigid objects, but obviously also very limited.

On the contrary, the strong mbb criterion can be met for the class of spatio-temporal primitive objects. The straightforward approach treats the bounding faces of an mbb as linear functions of time, similar to the bounding boxes in the indexing approach of Saltenis et al. (2000). The approximation scheme can be described as follows. Given a spatio-temporal object, its approximation is an mbb with time-dependent bounding faces. Through the linearity of the spatio-temporal objects (linear vertex movement and linear objects for each instant in time) it suffices to have piecewise linear functions for the bounding faces and to base the computations of these functions on the vertices being part of the spatio-temporal object. The motion function for the lower side of the x-dimension is defined as follows.

$$x^-(o) \quad := \quad \begin{cases} \min\limits_t\{p_x(t) \mid p \in \text{vertices}(o)\} & \text{for } t \in \text{time}(o) \\ \textbf{undef} & \text{for } t \notin \text{time}(o) \end{cases} \tag{6.1}$$

The remaining bounding faces are defined analogously (replace min by max for the upper boundaries of each face). For the spatio-temporal objects defined in this thesis, the motion functions are piecewise linear. Together, the definition of the six motion functions (two for each dimension) forms the *tmbb approximation scheme* (see also figure 6.2).

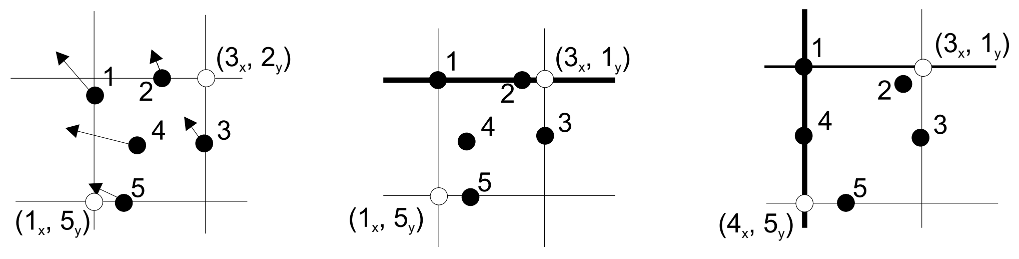


Figure 6.2: A tmbb (unfilled dots) defined by a set of temporal points (filled dots). *Left snapshot*: Vectors indicate movement (length means speed). *Middle snapshot*: Motion function of the tmbb upper right point changes (y-coordinate now defined by point 1). *Right snapshot*: Motion function of the tmbb lower left point changes (x-coordinate now defined by point 4).

6.1.4. Definition. (*temporal minimum bounding box, tmbb*) Let o denote a spatio-temporal object. The temporal minimum bounding box of o is a pair of temporal points

$$\text{tmbb} := [v^-,\ v^+]$$

where v^- and v^+ are called the lower left and the upper right point, respectively, and are defined by the linear motion functions defined in equation 6.1.

$$\begin{aligned} v^- &= (x^-(o),\ y^-(o),\ z^-(o)),\ \text{and} \\ v^+ &= (x^+(o),\ y^+(o),\ z^+(o)) \end{aligned}$$

For an instant $t \in \text{time}(o)$, a three-dimensional minimum bounding box of o is given by

$$\text{tmbb}(t) := [v^-(t),\ v^+(t)]$$

By definition, the following properties hold. First, v^-.lifetime equals v^+.lifetime. Second, for each $t \in v^-$.lifetime and each dimension i: $v_i^-(t) \leq v_i^+(t)$.

6.1.5. Proposition. *The tmbb approximation scheme for spatio-temporal objects meets the strong mbb criterion for the subset of spatio-temporal* primitive *objects.*

Proof: From the definition of a tmmb $b = [v^-, v^+]$ for a spatio-temporal object o it is obvious that for each $t \in \mathbb{T}$, $b(t) = \text{mbb}(o(t))$. It remains to show that a tmbb uses constant size for a spatio-temporal primitive object. But this follows immediately from the fact that these objects are temporal *simplices*, the vertices of which are linear functions of time. Then, the number of vertices is limited by a constant. Finally, the number of gsteps of v^- and v^+ is also limited by a constant. This fact follows also from the linearity of movement: if a motion function is based on a given vertex $v \in o$ for a time interval, and for a subsequent time interval it is based on a vertex $v' \in o$, then the motion function cannot be based on v again. □

The computation of each motion function can be performed iteratively over the vertices of a spatio-temporal object. In the following, the computation is described for the x^--function;

the remaining functions are treated similarly. The computation of the x^--function is initialised with the function that is undefined everywhere. The result of the computation is a time-line that associates a linear motion function with each interval on the time-line that together form the function for x^-. Then, assuming for the first $i-1$ vertices the time-line and its functions have been computed, the ith vertex v is added as follows. The time-line of v with its associated functions is built and merged with the computed time-line. After that each interval on the merged time-line is associated with two linear functions f, f_v that must be processed further to complete the merge. One of three cases can occur. First, both functions are equal (including two undefined functions). In this case, an arbitrary function can be chosen. Second, exactly one of the two functions for a given interval on the merged time-line is undefined. In this case, the defined function is chosen. Thirdly, both f and f_v are defined on the given interval I (and are not equal, which is case 1). In this case, either $f(t) \leq f_v(t)$ (or vice versa) for all $t \in I$, or I must be split into I_1 and I_2. In the former case, the smaller function is chosen. In the latter case, the split-instant is given by $t \in I$ such that $f(t) = f_v(t)$ and is unique (as the functions are linear and not equal). Then, for I_1 and I_2 the former case holds and the appropriate function can be chosen. Having merged both time-lines in this way, a post-processing step removes all unnecessary instants from the time-line, which are those whose neighbouring intervals are associated with the same function.

As the current chapter concentrates on intersects- and distance-based functions, the remainder of the section introduces two filter operations on tmbbs. First, it is shown how to detect intersections between two tmbbs. Second, it is shown how to compute a distance-based filter that is passed if two tmbbs have a distance less than a given threshold.

Intersection-detection for tmbbs. In the atemporal case, two mbbs b_1, b_2 intersect if and only if all of their x-, y, and z-interval pairs intersect. For simplicity, only the x-dimension is shown, as the remaining dimensions are handled equivalently. Let x_1^- denote the lower boundary and x_1^+ the upper boundary of the x-interval of b_1. The respective interval boundaries for b_2 are denoted by x_2^-, x_2^+. Let

$$\begin{aligned} v_x &:= x_1^- - x_2^+ \\ w_x &:= x_1^+ - x_2^- \end{aligned}$$

One can observe that the x-intervals intersect if and only if: (1) $v_x = 0$ or $w_x = 0$; or (2) $sgn(v_x) \neq sgn(w_x)$. On the whole, the bounding boxes intersect if and only if the x-, y, and z-intervals intersect.

For the temporal case, one iterates over the intervals on the merged time-line of two tmbbs b_1, b_2. Each such interval I has the property that the boundaries of the tmbbs are linear functions of time: $x_i^{+/-} = mt + c$. One obtains

$$\begin{aligned} v_x(t) &:= x_1^-(t) - x_2^+(t) \\ w_x(t) &:= x_1^+(t) - x_2^-(t) \end{aligned}$$

The x-intervals intersect at each instant in the temporal element

$$e_x := \{t \in \text{time}(b_1) \cap \text{time}(b_2) \mid v_x(t) = 0 \ \vee \ w_x(t) = 0 \ \vee \ sgn(v_x(t)) \neq sgn(w_x(t))\}$$

The temporal elements e_y and e_z are defined analogously. Then the tmbbs b_1, b_2 intersect at $e_x \cap e_y \cap e_z$. The construction of e_x is performed as follows. First, the merged time-line of

b_1 and b_2 is augmented by those instants t at which $sgn(v_x)$ or $sgn(w_x)$ changes its value. On this time-line, each interval I has the property that $sgn(v_x)$ and $sgn(w_x)$ are constant within I (excluding the boundaries of I). Starting with an empty e_x, an interval I of the time-line is inserted into e_x, if (1) $sgn(v_x) \neq sgn(w_x)$ holds on I, or if (2) $v_x = 0$ or $w_x = 0$ holds on I. (The boundaries of I must be handled properly.) Note that the instants on the time-line are degenerate intervals and must also be considered.

Distance-filter for tmbbs. The distance filter checks whether two tmbbs have a distance less than a (positive) threshold h for an instant in their lifetime. For atemporal mbbs b_1, b_2, the minimum Euclidean distance is given by

$$\begin{aligned} d(b_1, b_2) &= \sqrt{u_x^2 + u_y^2 + u_z^2} \quad \text{where} \\ u_i &= \min\{|i_1^- - i_2^+|,\ |i_1^+ - i_2^-|\} \end{aligned}$$

For the temporal case, one iterates over the intervals on the merged time-line of two tmbbs b_1, b_2. Each such interval I is processed as follows. The temporal variant of the u_i-expressions are defined as follows.

$$u_i(t) = \min_t\{|i_1^-(t) - i_2^+(t)|,\ |i_1^+(t) - i_2^-(t)|\}$$

The minimum now ranges over all t and therefore I must be split into several parts, where the splits are determined by the $u_i(t)$. The corresponding time-instant, if present and contained in I, is such a split-point. For each split-interval I', the function

$$d(b_1, b_2)(t) = \sqrt{u_x^2(t) + u_y^2(t) + u_z^2(t)}$$

can be searched for the minimum within I. As the square root can be omitted for the minimum search, one obtains a polynomial $at^2 + bt + c$ that contains in a, b, c the constants for the particular movement of the tmbb-boundaries. Using the first derivative, one can therefore obtain the solution by substituting these constants into $t = \frac{-b}{2a}$. One must check the boundaries of I' by evaluating the polynomial at the instants I'^- and I'^+. The global minimum on I' can thus be found. If for any of these intervals the distance is less than the given threshold h, the test is passed.

6.2 Operations for Spatio-Temporal Primitive Objects

This section investigates how to realise range queries, intersection-related operations, distance-related operations, and the restrictions on spatio-temporal primitive objects (see definition 6.1.1). Throughout this section it is assumed that the lifetime of a spatio-temporal primitive object equals the interval $[0, 1]$, which can be obtained by transforming the lifetime by translation and scaling[1].

6.2.1 Spatio-Temporal Range Queries

In the range queries to be presented, spatio-temporal objects can be tested for intersection with (atemporal) planes, half-spaces, and axis-aligned bounding boxes. Two different flavours

[1]This holds except for the distinction of open and closed intervals that must be handled accordingly.

of operations are distinguished: (1) a *boolean* operation yielding true if and only if an instant in time exists for that a primitive object intersected the query object; (2) a *temporal* operation yielding those temporal intervals during which a primitive object intersected the query object.

Intersection with a Plane

It is well-known that intersection is best expressed if one object is given in its implicit form and the other object in its parametric form (see, e.g., Abramowski and Müller, 1991). Intersecting spatio-temporal primitive objects with a plane, it is observed that it suffices to consider only the xy-plane, because the general case can be handled by applying an affine transform. This transform applies to every vector involved and it must map the given plane onto the xy-plane. Note, however, that only for presentational reasons this simplification has been chosen, which an implementation should avoid, since the affine transform will in general contain rotations—an unnecessary source of round-off errors. Secondly, the *implicit representation* of the plane is chosen

$$P_{xy}(x, y, z) = 0 \qquad :\Leftrightarrow \qquad z = 0$$

Then, in order to intersect P_{xy} with a parametric function $f(t) = (f_x(t), f_y(t), f_z(t))$ one can simply apply substitution, resulting in a single equation only.

$$P_{xy}(f_x(t), f_y(t), f_z(t)) = 0 \qquad \Leftrightarrow \qquad f_z(t) = 0$$

In the following, the problem of intersecting a temporal simplex with P_{xy} is investigated.

Plane with temporal point. The first object under consideration is a temporal point. Let $f \equiv \mathbf{p}$ be a temporal point that moves from location $\mathbf{a}$ to location $\mathbf{b}$:

$$\begin{array}{rcl} \mathbf{p} : (0,1) & \rightarrow & \mathbb{R}^3 \\ t & \mapsto & \mathbf{a} + t(\mathbf{b} - \mathbf{a}) \end{array}$$

Substitution yields

$$P_{xy}(\mathbf{p}(t)) = p_z(t) = a_z + t(b_z - a_z) = 0$$

$$t_0 = \frac{-a_z}{b_z - a_z} \tag{6.2}$$

The special case $a_z = b_z$ means movement parallel to the plane and the intersection is either empty or equal to $\mathbf{p}$. If $t_0 \in (0,1)$, the result is $\mathbf{p}(t_0)$, otherwise the intersection is empty. One immediately obtains:

$$\text{intersects}_{P_{xy}}(\mathbf{p}_{ab}) \equiv \begin{cases} t_0 \in (0,1) & \text{if } a_z \neq b_z \\ a_z = b_z = 0 & \text{if } a_z = b_z \end{cases}$$

$$\text{when-intersects}_{P_{xy}}(\mathbf{p}_{ab}) \equiv \begin{cases} (0,1) & \text{if } a_z = b_z = 0 \\ [t_0, t_0] & \text{if } t_0 \text{ exists and } t_0 \in (0,1) \\ \emptyset & \text{otherwise} \end{cases}$$

Plane with temporal segment. The next object under consideration is a temporal segment. Let $f \equiv \mathbf{s}$ be a temporal segment, based on two temporal points $\mathbf{p}(t) = \mathbf{a} + t\,(\mathbf{b} - \mathbf{a})$ and $\mathbf{q}(t) = \mathbf{c} + t\,(\mathbf{d} - \mathbf{c})$:

$$\begin{array}{rcl} \mathbf{s} : (0,1) \times [0,1] & \to & \mathbb{R}^3 \\ (t, \lambda) & \mapsto & \mathbf{p}(t) + \lambda\,(\mathbf{q}(t) - \mathbf{p}(t)) \end{array}$$

One observes that a segment intersects a plane if and only if one end-point is on one side of the plane and the other is on the other side of the plane, or if one of the end points lies within the plane. With respect to P_{xy} and to time, the following inequalities express the observation:

$$\begin{array}{rr} \text{I. } p_z(t) > 0 \ \wedge \ q_z(t) < 0 & \text{II. } p_z(t) < 0 \ \wedge \ q_z(t) > 0 \\ \text{III. } p_z(t) = 0 & \text{IV. } q_z(t) = 0 \end{array}$$

Let S_I, S_{II}, S_{III}, and S_{IV} denote the solution sets for the respective linear inequality (or system of linear inequalities). Then, the instants in the following solution set S correspond exactly to the instants at which the temporal segment intersects P_{xy}:

$$S := (0,1) \cap \bigcup_{i=I..IV} S_i$$

Hence, the intersects and when-intersects operations can be defined as follows.

$$\begin{array}{rcl} \text{intersects}_{P_{xy}}(\mathbf{s}) & \equiv & S \neq \emptyset \\ \text{when-intersects}_{P_{xy}}(\mathbf{s}) & \equiv & S \end{array}$$

The exact computation of the solution sets is straightforward and therefore omitted here.

Plane with temporal triangle. The next object under consideration is a temporal triangle. Let $\mathbf{tr}$ be a temporal triangle to be intersected with P_{xy}, based on the three temporal vertices $\mathbf{p}(t) = \mathbf{a} + t\,(\mathbf{b} - \mathbf{a})$, $\mathbf{q}(t) = \mathbf{c} + t\,(\mathbf{d} - \mathbf{c})$, $\mathbf{r}(t) = \mathbf{e} + t\,(\mathbf{f} - \mathbf{e})$:

$$\begin{array}{rcl} \mathbf{tr} : (0,1) \times [0:1] \times [0,1] & \to & \mathbb{R}^3 \\ (t, \lambda, \kappa) & \mapsto & \mathbf{p}(t) + \lambda\,(\mathbf{q}(t) - \mathbf{p}(t)) + \kappa\,(\mathbf{r}(t) - \mathbf{p}(t)),\ \lambda + \kappa \leq 1 \end{array}$$

This case can be conveniently reduced to the case of temporal segments, since due to the linearity, the interior of a triangle cannot intersect P_{xy} without its boundary intersecting P_{xy}. Let $\overline{\mathbf{pq}}, \overline{\mathbf{pr}}, \overline{\mathbf{rq}}$ denote the temporal segments that form for each $t \in [0,1]$ the boundary of $\mathbf{tr}(t)$.

$$\begin{array}{rcl} \text{intersects}_{P_{xy}}(\mathbf{tr}) & \equiv & \bigvee_{\mathbf{s} \in \{\overline{\mathbf{pq}}, \overline{\mathbf{pr}}, \overline{\mathbf{rq}}\}} \text{intersects}_{P_{xy}}(\mathbf{s}) \\ \text{when-intersects}_{P_{xy}}(\mathbf{tr}) & \equiv & \bigcup_{\mathbf{s} \in \{\overline{\mathbf{pq}}, \overline{\mathbf{pr}}, \overline{\mathbf{rq}}\}} \text{when-intersects}_{P_{xy}}(\mathbf{s}) \end{array}$$

Plane with temporal tetrahedron. The case of a temporal tetrahedron can be handled similar to the previous case. The operations “intersects” and “when-intersects” can be reduced to the bounding temporal triangles of the temporal tetrahedron.

Intersection with a Half-Space

Again, the discussion can be limited to the case, where the half-space is bounded by P_{xy}. Hence, for the remainder of the section, objects are intersected with $H_{xy} := z \geq 0$.

Half-space with temporal point. The following definitions make use of t_0 (see equation 6.2), denoting the instant at which the temporal point intersects plane P_{xy}. Let the temporal point move from $\mathbf{a} \in \mathbb{R}^3$ to $\mathbf{b} \in \mathbb{R}^3$.

$$\begin{aligned}\text{intersects}_{H_{xy}}(\mathbf{p}) &\equiv a_z \geq 0 \ \vee \ b_z \geq 0 \\ \text{when-intersects}_{H_{xy}}(\mathbf{p}) &\equiv \begin{cases} [\max\{t_0, 0\}, 1] & \text{if } b_z > a_z \ \wedge \ b_z \geq 0 \\ [0, \min\{t_0, 1\}] & \text{if } a_z > b_z \ \wedge \ a_z \geq 0 \\ [0, 1] & \text{if } a_z = b_z \ \wedge \ a_z \geq 0 \\ \emptyset & \text{otherwise} \end{cases}\end{aligned}$$

Half-space with temporal segment. For a temporal segment $\mathbf{s}$, the intersection operations can be reduced to the operations on the temporal vertices of the segment. Let the temporal vertices of the temporal segment be $\mathbf{p}$ and $\mathbf{q}$.

$$\begin{aligned}\text{intersects}_{H_{xy}}(\mathbf{s}) &\equiv \text{intersects}_{H_{xy}}(\mathbf{p}) \ \vee \ \text{intersects}_{H_{xy}}(\mathbf{q}) \\ \text{when-intersects}_{H_{xy}}(\mathbf{s}) &\equiv \text{when-intersects}_{H_{xy}}(\mathbf{p}) \ \cup \ \text{when-intersects}_{H_{xy}}(\mathbf{q})\end{aligned}$$

Half-space with temporal triangle/tetrahedron. As with temporal segments, the intersection operations for temporal triangles and tetrahedra can be reduced to their temporal vertices. The definition of the operations is therefore a simple extension of the operations for temporal segments.

Intersection with a Bounding Box

In the sequel, it is assumed that the spatio-temporal primitive objects under investigation are to be intersected with the 4D query box for the whole validity interval $(0, 1)$ of the spatio-temporal primitive object. Hence, the query box is 3D. A query box is aligned to the coordinate axes. The 3D query box is the intersection of the six half spaces $H_1 := x \geq c_1$, $H_2 := y \geq c_2$, $H_3 := z \geq c_3$, $H_4 := x \leq c_4$, $H_5 := y \leq c_5$, $H_6 := z \leq c_6$:

$$B \ := \ \bigcap_{i=1..6} H_i$$

Box with temporal point. Since the intersects operations for a temporal point $\mathbf{p}$ and a half space have been derived, one immediately obtains:

$$\begin{aligned}\text{intersects}_B(\mathbf{p}) &\equiv \bigcap_{i=1..6} \text{when-intersects}_{H_i}(\mathbf{p}) \neq \emptyset \\ \text{when-intersects}_B(\mathbf{p}) &\equiv \bigcap_{i=1..6} \text{when-intersects}_{H_i}(\mathbf{p})\end{aligned}$$

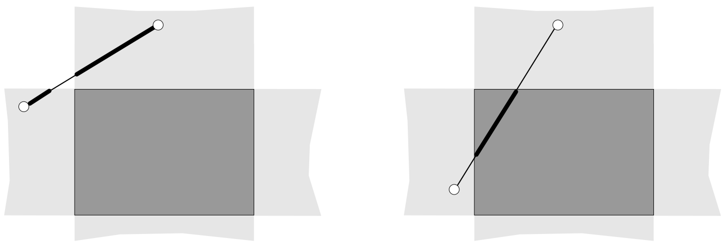


Figure 6.3: (Atemporal case) Left: Although the segment intersects all half-spaces of the query box, one cannot conclude that the segment also intersects the query box. Right: A segment intersects the query box if the intersecting parts of the segment have points in common.

Box with temporal segment. The intersects operation of a temporal segment with a query box is more involved, since it cannot be trivially reduced to the intersections with a half-space (see also figure 6.3). The right part of figure 6.3 indicates a strategy that can be extended to the temporal case. In brief (neglecting parallel arrangements), for an atemporal segment $\mathbf{s} = \overline{\mathbf{pq}}$ one computes six values λ_i for each half-space H_i such that $\mathbf{p} + \lambda_i \cdot (\mathbf{q} - \mathbf{p})$ intersects the bounding plane of half-space H_i. λ_i determines a solution set S_i: for each $\lambda^* \in S_i$ the 3D-point

$$\mathbf{p} + \lambda^* \cdot (\mathbf{q} - \mathbf{p})$$

is located within H_i. If $\lambda_i \notin [0,1]$, then $S_i = \emptyset$ or $S_i = [0,1]$, depending on location. Otherwise, $S_i = [0, \lambda_i]$ or $S_i = [\lambda_i, 1]$, depending on orientation. The solution set for the whole query box B is therefore

$$S \;:=\; \bigcap_{i=1..6} S_i \tag{6.3}$$

The segment intersects the query box if and only if $S \neq \emptyset$.

In the case, where $\mathbf{s}(t) = \overline{\mathbf{p}(t)\mathbf{q}(t)}$ is a temporal segment, each half-space is associated with a function $\lambda_i : \mathbb{R} \to \mathbb{R}$ describing the intersection with the bounding plane of half-space H_i. Function λ_i is given by:

$$\lambda_i(t) \;=\; c_i - \frac{p_i(t)}{q_i(t) - p_i(t)} \tag{6.4}$$

Note that λ_i is a certain rational function $f(x) = \frac{P(x)}{Q(x)}$, where $P(x)$ has degree ≤ 1 and $Q(x)$ has degree $= 1$. The point

$$\mathbf{p}(t) + \lambda_i(t) \cdot (\mathbf{q}(t) - \mathbf{p}(t))$$

intersects plane P_i (for each t for which λ_i is defined). The solution set $S_i \subseteq [0,1] \times [0,1]$ is the set of (t^*, λ^*)-pairs for which an intersection with H_i results:

$$S_i := \{(t^*, \lambda^*) \mid \mathbf{p}(t^*) + \lambda^* \cdot (\mathbf{q}(t^*) - \mathbf{p}(t^*)) \in H_i\}$$

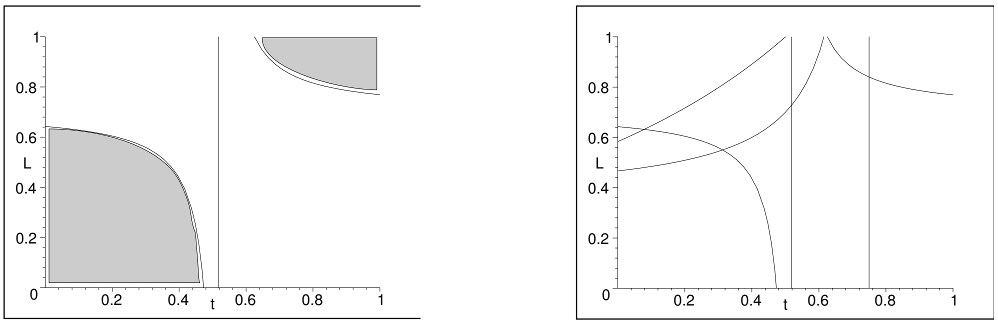


Figure 6.4: The graph of λ_i is the boundary of the shaded region describing the solution set S_i. The asymptote (if existing) marks the instant at which the region switches from below-the-graph to above-the-graph, or vice-versa.

The solution set for the whole query box is defined as in equation 6.3. The "intersects"-operation returns true if and only if $S \neq \emptyset$. The "when-intersects"-operation returns S_t, the projection of S onto the t-axis. In the following, the focus is on how to compute S.

In summary, the procedure constructs the region in $[0,1] \times [0,1]$ that corresponds to S (see also figure 6.4) and projects it onto the time-axis. The region is enclosed by two chains, which consist of parts of the λ_i-graphs: an upper chain, which bounds the region from above, and a lower chain, which bounds the region from below. Hence, the procedure works in three steps: (1) it constructs the upper and the lower chains; (2) it combines them to form the solution set; and (3) it projects the region onto the t-axis and returns the corresponding temporal element.

However, the computation must deal with the degenerate configuration that λ_i is undefined for all t. Function λ_i is undefined for all t with $q_i(t) = p_i(t)$ (see equation 6.4). The following cases can be distinguished:

1. for all t: $q_i(t) = p_i(t) \neq 0$; geometric interpretation: $\mathbf{s}(t)$ is parallel to, but not contained in P_i.

2. for all t: $q_i(t) = p_i(t) = 0$; geometric interpretation: like 1., but $\mathbf{s}(t)$ is contained in P_i.

If for a λ_i case 1 holds and the segment does not intersect H_i, the computation stops with the particular return value ("**false**" for operation intersects and "$\emptyset$" for operation when-intersects). Otherwise, in both cases the solution set S_i equals $[0,1] \times [0,1]$. Therefore, H_i does not "contribute" to S and can be excluded from further examination. In the sequel, it is therefore assumed that the functions λ_i are not degenerate.

It follows that there is at most one t_i with $q_i(t_i) = p_i(t_i)$. The instant t_i is important for the solution set S_i: depending on the orientation of the segment, at t_i the solution set "switches" from the region *above* the graph of λ_i to the region *below* the graph of λ_i, or vice-versa (see also figure 6.4). Furthermore, each function λ_i is split in two parts λ_i^L, λ_i^U, corresponding to the lower boundary and the upper boundary of the solution set, respectively.

The dividing instant is the asymptote of λ_i, or if it does not exist, one of the two functions is undefined. The decision, which side of the asymptote is which boundary, can be based on simply probing the function, for instance, at $t^* < t_i$ to obtain a (t^*, λ^*)-pair and checking whether the 3D-point

$$\mathbf{v} = \mathbf{p}(t^*) + (\lambda^* - \delta) \cdot (\mathbf{q}(t^*) - \mathbf{p}(t^*))$$

lies within H_i for an appropriately sized $\delta > 0$. With these choices for t^* and δ one checks the left side of the asymptote below the graph of λ_i. Hence, if $\mathbf{v}$ lies within H_i, the left side of the asymptote is λ_i^U, while the right side is λ_i^L, and vice-versa.

Additionally, the solution set S for the whole query box B is bounded by 0 and 1 for both parameters t and λ. Therefore, the functions are defined such that

$$\begin{aligned}
\lambda_i^L, \lambda_i^U &: [0,1] \to \mathbb{R} \\
\lambda_i^L(t) &= \begin{cases} \lambda_i(t) & \text{if } \lambda_i(t) \in [0,1] \text{ and } \mathbf{s}(t, \lambda_i(t) + \delta) \in H_i,\ \delta > 0 \\ 0 & \text{if } \lambda_i(t) < 0 \\ \text{undefined} & \text{otherwise} \end{cases} \\
\lambda_i^U(t) &= \begin{cases} \lambda_i(t) & \text{if } \lambda_i(t) \in [0,1] \text{ and } \mathbf{s}(t, \lambda_i(t) - \delta) \in H_i,\ \delta > 0 \\ 1 & \text{if } \lambda_i(t) > 1 \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

On this basis, two chains can be defined for the solution set of the whole query box B. The chains are functions such that:

$$\begin{aligned}
LC(t) &= \begin{cases} \max_{i=1,\dots,6}\{\lambda_i(t)\} & \text{if } \lambda_i^L(t) \text{ is defined for all } i \\ \text{undefined} & \text{otherwise} \end{cases} \\
UC(t) &= \begin{cases} \min_{i=1,\dots,6}\{\lambda_i(t)\} & \text{if } \lambda_i^U(t) \text{ is defined for all } i \\ \text{undefined} & \text{otherwise} \end{cases}
\end{aligned}$$

Then, the solution set S for the query box B is given by the (t, λ)-pairs enclosed within the upper chain UC and the lower chain LC. The algorithms 6.2.1 and 6.2.2 compute both the upper and the lower chain in an iterative manner. A chain is represented by a set of intervals in ascending t-order. Each interval is accompanied by the function λ_i whose graph forms the boundary of the solution set during this interval. Algorithm 6.2.1 computes first the functions λ_i^L and λ_i^U (lines 1–7) and stores them as chains for H_i. The detailed computation is omitted here, but should be straightforward from the above discussions. The lower and upper chains (LC and UC) are initialised with the interval $[0,1]$ and the constant functions $t \mapsto 0$ and $t \mapsto 1$, respectively (lines 8 and 9). In lines 10–16 the chains for λ_i are merged with the current chains LC and UC. The procedure for the merging the lower chains is depicted in algorithm 6.2.2. Merging of upper chains is omitted here, but is similar to merging lower chains (replace "max" by "min" in line 10).

Procedure "merge_lower_chains" works as follows. It merges the two lower chains C_1 and C_2 into the new chain C. To this end, the procedure iterates through both input chains in ascending t-order by popping intervals from the chains into the variables I_1 and I_2. Only if they overlap, it processes I_1 and I_2. The iteration through the chains is managed in the lines 5, 6, 8, 12, and 13. A pair of overlapping intervals is processed as follows. Importantly, in line 9 the intersection of the associated functions—limited to the overlapping region of the intervals—are computed. By the nature of these functions, at most two intersection points are possible (assuming the functions are not equal). The overlapping interval is split into several intervals for these intersection points (line 10, for presentational reasons assuming two intersections exist). Each resulting interval j is associated with one of the two functions associated with the intervals I_1 and I_2, choosing the function that yields bigger function values on j.

Finally, in algorithm 6.2.1 (line 17), procedure "get_domain" returns the set S_t of instants for which the upper chain UC is above or equal to the lower chain LC. The pseudo-code for procedure "get_domain" is omitted, as it is similar to the merging of chains. It also iterates over intersecting intervals of the two chains. Each such intersection of an interval-pair is processed as follows. First, the intersection points of the two associated functions are calculated, splitting the interval into sub-intervals. Each sub-interval is tested, if the function of the upper chain (for this interval) is bigger than the function of the lower chain. If this holds, the sub-interval must be inserted into the resulting temporal element. After checking each interval-pair, the resulting temporal element contains exactly the instants at which the segments intersects the query box.

Algorithm 6.2.1 segment-box-intersection(s, $B = \cap_{i=1,\ldots,6} H_i$)

```
1:  for $i = 1, \ldots, 6$ do
2:    $H_i$.LC $\Leftarrow$ compute $\lambda_i^L$ for $s$
3:    $H_i$.UC $\Leftarrow$ compute $\lambda_i^U$ for $s$
4:    if $H_i$.LC $= \emptyset$ or $H_i$.UC $= \emptyset$ then
5:      return empty intersection
6:    end if
7:  end for
8:  LC $\Leftarrow [[0, 1], t \mapsto 0]$
9:  UC $\Leftarrow [[0, 1], t \mapsto 1]$
10: for $i = 1, \ldots, 6$ do
11:   LC $\Leftarrow$ merge_lower_chains(LC, $H_i$.LC)
12:   UC $\Leftarrow$ merge_upper_chains(UC, $H_i$.UC)
13:   if LC $= \emptyset$ or UC $= \emptyset$ then
14:     return empty intersection
15:   end if
16: end for
17: return get_domain(LC, UC)
```

Box with temporal triangle/tetrahedron. While the presented approach for box with a temporal segment is advantageous, as it also allows to approximate the intersection of the two objects, with triangles and tetrahedra one and two additional parameters are introduced, respectively. The approach becomes therefore more complex and a simpler, but practicable approach is justified. It is based on the distance-function that computes the minimum Euclidean distance between two spatio-temporal objects, which is presented subsequently. The faces of the query box can be triangulated and, using the distance-function, the minimum distance of the spatio-temporal primitive object (triangle/tetrahedron) to the faces of the box is calculated (or *when* the minimum distance is reached, for the temporal function). The

Algorithm 6.2.2 merge_lower_chains(C_1, C_2)

```
1: $I_1 \Leftarrow [1,0]$
2: $I_2 \Leftarrow [1,0]$
3: $C \Leftarrow \emptyset$
4: while $C_1 \neq \emptyset$ and $C_2 \neq \emptyset$ do
5:     while $I_1^+ < I_2^-$ and $C_1 \neq \emptyset$ do $I_1 \Leftarrow \text{pop}(C_1)$
6:     while $I_2^+ < I_1^-$ and $C_2 \neq \emptyset$ do $I_2 \Leftarrow \text{pop}(C_2)$
7:     $I \Leftarrow [\max\{I_1^-, I_2^-\}, \min\{I_1^+, I_2^+\}]$
8:     if $I^- > I^+$ then return $C$
9:     $t, t' \Leftarrow$ intersection-points($I_1.f_{|[I^-,I^+]}, I_2.f_{|[I^-,I^+]}$)
10:    insert intervals $[I^-, t]$, $[t, t']$, $[t', I^+]$ into $C$
11:    for each inserted interval $j$ do $j.\mathrm{f} \Leftarrow \max\{I_1.f_{|j}, I_2.f_{|j}\}$
12:    if $I_1^+ \leq I_2^+$ then $I_1 \Leftarrow \text{pop}(C_1)$
13:    if $I_2^+ \leq I_1^+$ then $I_2 \Leftarrow \text{pop}(C_2)$
14: end while
15: return $C$
```

result must be combined with a further computation: the case, when the spatio-temporal object does not intersect the faces of the query box, but its interior; however, this case can be reduced to the case of when all of the object‘s temporal vertices are contained in the query box.

6.2.2 Euclidean Metric Operations

Two different flavours of the Euclidean metric operation are developed: (1) an operation that reports the minimum distance between two spatio-temporal objects and the instant (or interval) in time when that happened; (2) an operation that reports the instants at which the distance between objects have been minimal. Again, the mathematical functions are described first. The former operation then corresponds to an extreme value problem, and it is shown how to solve this problem.

Since the dimension of temporal simplices is limited, it is possible to implement specialised algorithms, one for each possible temporal simplex combination; however, these implementations can be described in a general framework.

1. Problem. *Let s_1, s_2 be two spatio-temporal primitive objects. The minimum Euclidean distance problem for s_1, s_2 is defined as*

$$\textit{find the value } \min_{t\in[0,1]} \{ \ d(s_1(t), s_2(t)) \ \}$$

To solve the minimum distance problem on two given arbitrary spatio-temporal primitive objects s_1 and s_2, it is not sufficient to compute $\min\{d(s_1(0), s_2(0)), d(s_1(1), s_2(1))\}$, since the minimum may be reached for $t \in (0,1)$. The key to the solution lies in parameterising the 3D Euclidean distance formula

$$\sqrt{\sum_{i=1}^{3} (\mathbf{x}_i - \mathbf{y}_i)^2} \tag{6.5}$$

valid for two 3D points $\mathbf{x}$ and $\mathbf{y}$, to be described below. If then the number of parameters equals p, the resulting expression can be viewed as a function $dist : \mathbb{R}^p \to \mathbb{R}$. Therefore, calculus methods can be applied to solve the problem of computing the minimum distance. Hence, function *dist* is analysed for local minima and to this end the (non-linear) system of equations

$$\frac{\partial dist}{\partial \lambda_1} = 0, \ldots, \frac{\partial dist}{\partial \lambda_p} = 0 \tag{6.6}$$

is built, which must be solved at least for one variable.

Beforehand, one must be sure that *dist* is a differentiable function, otherwise the approach is not going to work. To this end, the parameterisation of formula 6.5 is described. First, there are zero or more "spatial" parameters which give point coordinates of all points of a purely spatial simplex. In particular, a sole 3D point has no such parameter, whereas a segment $\mathbf{s} = \overline{\mathbf{pq}}$ has one parameter, and a triangle $\mathbf{t} = \Delta(\mathbf{p}, \mathbf{q}, \mathbf{r})$ has two, since

$$\begin{aligned} \mathbf{s} &= \mathbf{p} + \lambda \cdot (\mathbf{q} - \mathbf{p}), \quad 0 \leq \lambda \leq 1 \\ \mathbf{t} &= \mathbf{p} + \lambda_1 \cdot (\mathbf{q} - \mathbf{p}) + \lambda_2 \cdot (\mathbf{r} - \mathbf{p}), \\ & \qquad \lambda_i \geq 0, \ \lambda_1 + \lambda_2 \leq 1 \end{aligned}$$

This results in parameterisation of a simplex at a given, but fixed instant.

Secondly, for the instants in $(0, 1)$ there is one temporal parameter. The vertices $\mathbf{p}_i$ of a temporal simplex are already expressed in terms of time (see above), and their trajectories are at hand.

$$\mathbf{p}_i(t) = \mathbf{p}_i(0) + t \cdot (\mathbf{p}_i(1) - \mathbf{p}_i(0)), \ 0 < t < 1$$

The points in a d-dimensional temporal simplex $\mathbf{s}$ having temporal points $\mathbf{p}_1, \ldots, \mathbf{p}_{d+1}$ the coordinates of a point contained in s are expressed as

$$s(t, \lambda_1, \ldots, \lambda_d) = \mathbf{p}_1(t) + \sum_{i=2}^{d+1} \lambda_i \cdot (\mathbf{p}_i(t) - \mathbf{p}_1(t))$$

Finally, given a d_1-dimensional simplex s and a d_2-dimensional simplex s', formula 6.5 can be rewritten as $dist(.)$:

$$\sqrt{\sum_{i=1}^{3} (s_i(t, \lambda_1, \ldots, \lambda_{d_1}) - s'_i(t, \kappa_1, \ldots, \kappa_{d_2}))^2} \tag{6.7}$$

Note that the time parameter t is shared by the functions s_i and s'_i.

To prove that $dist(.)$ is differentiable, it now suffices to state that it is a composition of differentiable functions. Consequently, in order to find the global minimum one can firstly seek local minima by differentiating *dist*. It is important to note that the domain of *dist* is restricted. It consists of all "valid" parameter configurations. The boundary of this domain $\mathcal{D}$ is given by the formulas

$$\begin{aligned} t &\in \{0, 1\} && (6.8) \\ \lambda_i &= 0, \ \forall i \in \{1, \ldots, d_1\} && (6.9) \\ \kappa_i &= 0, \ \forall i \in \{1, \ldots, d_2\} && (6.10) \\ \sum_{i=1}^{d_1} \lambda_i &= 1 && (6.11) \\ \sum_{i=1}^{d_2} \kappa_i &= 1 && (6.12) \end{aligned}$$

The global minimum of $dist_{|\mathcal{D}}$ may not correspond to a local minimum of the unrestricted *dist*-function. In this case, the global minimum lies on the boundary of $\mathcal{D}$ which must therefore be investigated. Yet, this is solved easily. Fixing the time parameter (eq. 6.8) means computing distances between purely spatial simplices, whereas equations 6.9 to 6.12 in fact mean reduction of dimensionality of one simplex.

More concrete, formulating specific procedures for the distance computation for two given types of spatio-temporal primitive objects, consists of the following steps.

1. Parameterising formula 6.5 according to the dimension of the simplices (cf. formula 6.7) with a shared time parameter. The coordinates describing point-movement appear as constants; they are input into the procedure via the temporal simplices.
2. Partial differentiation of the resulting expression (cf. equations 6.6).
3. Solving this system of equations for the time parameter t, resulting in a univariate polynomial $p(t)$, the coefficients of which are expressions of the coordinates for point-movement[2].

On this basis, the procedure computing the minimum distance between two spatio-temporal primitive objects s, s' proceeds as follows.

1. Given the simplices in parameterised form, it evaluates the expressions for the coefficients of the polynomial $p(t)$.
2. Then, it finds the (real) roots of $p(t)$ within interval $(0, 1)$. Here, known numerical methods for root-finding can be employed. Using numerical methods here is justified, as polynomial $p(t)$ can assume degree five (temporal parameter t, two spatial parameters, for instance in segment/segment-distance) that cannot be solved analytically in general.
3. For each such root, that means time instant, the procedure computes the distance between $s(t)$ and $s'(t)$ and stores the resulting value.
4. The procedure checks the boundary of $\mathcal{D}$ by solving the problem of lower dimensionality, storing the resulting values.
5. Finally, the procedure returns the minimum value among those computed in steps 3 and 4.

Note that it is not necessary to check if at a root of $p(t)$ there is actually a local minimum[3], since in any case the global minimum on $\mathcal{D}$ is found.

The underlying space is $\mathbb{R}^3$. In conjunction with the fact that the above functions are not per se restricted to $\mathcal{D}$ and therefore at each instant operate on affine subspaces of $\mathbb{R}^3$, instead of convex hulls of the vertices involved, some simplifications become possible. For example, computing the distance between a temporal triangle T and a temporal segment S would at first stage reduce to distance between a plane and a line. This is not very useful,

[2]Note that the square root need not be considered.

[3]As opposed to a local maximum or a saddle point.

since—if not parallel—they are bound to have a distance of 0. Furthermore, if not parallel, then points that lie in the *interior* of T and in the interior of S generally do not have the minimum distance between both objects. The exception is the parallel case, but then one can find a point on one of the object's boundaries. In fact, one can immediately proceed to check the boundary $\mathcal{D}$. Here, this involves computing: (1) distances at the instants $t = 0, t = 1$; (2) distances of the temporal boundary segments of the triangle T to segment S; (3) distances of the bounding temporal points of S to triangle T.

6.2.3 Ensuring Object Restrictions

The restrictions for a spatio-temporal object must be checked on creation and update. Thus far, however, the computation of the geometric restrictions have been left open. In brief, geometric restrictions concern (1) temporal simplices: for the whole lifetime, a temporal segment may not degenerate to a single point, a temporal triangle may not be without area, a temporal tetrahedron may not be flat; and concern (2) temporal meshes: for the whole lifetime, the mesh-criterion may not be violated. In the following, both problems of simplex restrictions and mesh restrictions are solved.

2. Problem. *Let o be a spatio-temporal primitive d-simplex ($d = 1, \ldots, 3$). Check if there exists a $t \in (0, 1)$ at which o degenerates, that means its vertices are affine dependent.*

The problem can be solved straightforwardly for the atemporal case. The solution is then extended to the temporal case. In general, a tuple $(p_0, \ldots, p_n)$ of $n+1$ points is called affine independent, if the n vectors $p_1 - p_0$, $p_2 - p_0$, $\ldots$, $p_n - p_0$ are linearly independent. In the following, computations are derived for temporal segments, temporal triangles, and temporal tetrahedra.

In particular, two points $p_1, p_2 \in \mathbb{R}^3$ form a (non-degenerated) segment, if the vector $p_2 - p_1$ is linearly independent, that means, if it is not equal to the null vector. Extending to the temporal case, two temporal points are given as

$$\begin{aligned} \mathbf{p}_1(t) &= \mathbf{a} + t \cdot (\mathbf{b} - \mathbf{a}) \\ \mathbf{p}_2(t) &= \mathbf{c} + t \cdot (\mathbf{d} - \mathbf{c}) \end{aligned}$$

Checking for affine dependency means checking for linear dependency of the vector (after re-arranging and substitution):

$$\begin{aligned} \mathbf{p}_2(t) - \mathbf{p}_1(t) &= 0 \quad \Leftrightarrow \\ \mathbf{c} - \mathbf{a} + t \cdot (\mathbf{a} - \mathbf{c} + \mathbf{d} - \mathbf{b}) &= 0 \quad \Leftrightarrow \\ \mathbf{a}^* + t \cdot \mathbf{b}^* &= 0 \quad \Leftrightarrow \\ \begin{bmatrix} a_1^* \\ a_2^* \\ a_3^* \end{bmatrix} + t \cdot \begin{bmatrix} b_1^* \\ b_2^* \\ b_3^* \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \end{aligned} \tag{6.13}$$

Hence, it must be checked, if a t in $(0, 1)$ exists such that condition 6.13 holds. For these instants, the given two points $p_1(t), p_2(t)$ coincide and do not form a segment and therefore violate the *segment*-restriction. To obtain a procedure with parameters $\mathbf{a}, \mathbf{b}$ for $p_1(t)$ and $\mathbf{c}, \mathbf{d}$ for $p_2(t)$, the first stage is to compute the vectors $\mathbf{a^*}$ and $\mathbf{b^*}$. Then, for each equation $a_i^* + t \cdot b_i^* = 0$ the set of instants in $(0, 1)$ can be computed that solve the equation; possible solutions are $(0, 1)$, $\{t\}$ for a single $t \in (0, 1)$, and $\emptyset$. Finally, it is checked if the intersection of the solution sets are non-empty; in this case the temporal points violate the restrictions for a temporal segment.

Similarly, three points $p_1, p_2, p_3 \in \mathbb{R}^3$ form a degenerate triangle, if the two vectors $p_2 - p_1$ and $p_3 - p_1$ are linearly dependent. Put differently, the vector product of both vectors is then equal to the null vector. Extending to the temporal case, three temporal points $p_1(t), p_2(t), p_3(t)$ are given as above. Checking for affine dependency means checking for linear dependency. Expressed in terms of the cross product:

$$(p_2(t) - p_1(t)) \times (p_3(t) - p_1(t)) \quad = \quad 0$$

Substituting $u(t) = p_2(t) - p_1(t)$ and $v(t) = p_3(t) - p_1(t)$, one obtains

$$u(t) \times v(t) = \begin{bmatrix} u_2(t)v_3(t) - v_3(t)w_2(t) \\ v_3(t)w_1(t) - v_1(t)w_3(t) \\ v_1(t)w_2(t) - v_2(t)w_1(t) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The components of the cross product are polynomials in t of degree two, the roots of which can be found with well-known methods. Again, if the intersection of these results and the interval $(0, 1)$ is non-empty, the three temporal points do not form a valid triangle for these instants; in this case the restriction is violated.

Four points $p_1, p_2, p_3, p_4 \in \mathbb{R}^3$ form a degenerate tetrahedron, if the three vectors $u = p_2 - p_1$, $v = p_3 - p_1$, and $w = p_4 - p_1$ are linearly dependent. Put differently, the determinant of the matrix comprised of these three vectors is equal to 0. Extending to the temporal case, four temporal points $p_1(t), p_2(t), p_3(t), p_4(t)$ are given as above. Checking for affine dependency means checking for linear dependency. Expressed in terms of the determinant:

$$\begin{vmatrix} u_1(t) & u_2(t) & u_3(t) \\ v_1(t) & v_2(t) & v_3(t) \\ w_1(t) & w_2(t) & w_3(t) \end{vmatrix} = 0$$

When evaluated, the determinant is a polynomial in t of degree three, the roots of which can be found with well-known methods. Like before, if the result intersected with the interval $(0, 1)$ is non-empty, the given temporal points violate the tetrahedron restriction.

Extending these computations from spatio-temporal primitive objects to temporal simplices is simply done by iterating over the gsteps-intervals of the temporal simplex and perform the test for the induced primitive objects. If a test fails for any of the primitive object, the test fails for the temporal simplex. If otherwise none of the tests for the primitive objects fails, the temporal simplex passes the test.

The problem of checking for mesh restriction is described as follows.

3. PROBLEM. *Let $\mathcal{C} = [S, e]$ be a temporal mesh and $s' = [V, e']$ a temporal simplex with $s' \notin S$. Check if there exists an instant $t \in e'$ and a temporal simplex $s \in S$ such that $s'(t)$ intersects $s(t)$ and not $s'(t) \leftrightarrow s(t)$, that means the simplices intersect but are not neighbours.*

The problem can be solved as follows. Iterating over the temporal simplices of the mesh, each simplex s is tested against s'. This test iterates over the instants of the merged time-line for s and s', yielding two spatio-temporal primitive objects o_1, o_2 for each section of the merged time-line. Then, the set $N \subset [0,1]$ is computed for which instants t the relationship $o_1(t) \Leftrightarrow o_2(t)$ holds. Next, it is computed if the distance between o_1 and o_2 reaches zero for any $t \in [0,1] \setminus N$, using methods from the previous section. If no such t is found, the objects pass the test. Otherwise, the procedure reports an invalid intersection. Of course, an available index for the mesh can be exploited, as well as the geometric intersection filter described in section 6.1.3.

6.3 Operations for (Complete) Spatio-Temporal Objects

The previous section centred on operations for spatio-temporal primitive objects. These operations can be seen as base operations to be applied on spatio-temporal objects. Putting aside the restrictions for temporal simplices that have been described in the previous section, the operations under consideration are summarised in the following.

The operations operate on arbitrary objects of class *stObject*. Q denotes the type of a query object, which can be a plane, half-space, bounding box, or *stObject*.

Boolean operations:
intersects: $Q \rightarrow$ *bool*
disjoint: $Q \rightarrow$ *bool*

Temporal operations:
when_intersects: $Q \rightarrow$ *temporalElement*
when_disjoint: $Q \rightarrow$ *temporalElement*
when_mindist: $Q \rightarrow$ *temporalElement*

Metric operation:
min_dist: $Q \rightarrow$ *scalar*

Range Query:
range_query: $Q \rightarrow$ *stObject*

The meaning of the latter operation, performed on a spatio-temporal object o and a query object q, can be described as the extraction of the parts of o that intersect the query object q, limited to the temporal element at which the intersection happened:

$$o.\text{range\_query}(q) = \{s_{|e} \mid s \in o,\ e = s.\text{when\_intersects}(q)\}$$

As mentioned above, these operations are broken down to the level of spatio-temporal primitive objects. Then, the base operations from section 6.2 can be applied immediately. From the sub-results, the global result must be inferred. With this observation it is possible to describe the operations in a common framework, the so-called decomposable search problem (Bentley, 1979; Mehlhorn, 1984). Using available indexes for spatio-temporal objects, the "relevant" primitive objects can be found quickly.

The remainder of the section is therefore organised as follows. The next section introduces the decomposable search problems and how they generalise the above operations. The remaining sections focus on the different starting positions of index presence or index absence.

6.3.1 Decomposable Problems

It is possible to generalise operations like spatio-temporal distance or Boolean intersects. It can be observed that the result of, e.g., a spatio-temporal *distance* between two temporal meshes can be assembled by computing the distance between each primitive object-pair and afterwards combining these sub-results by the operation *min*. Here, both base computation and assembling operation have time complexity $O(1)$. Furthermore, it can be observed that the order of these sub-computations is unimportant for the result. More generally, let $\mathcal{C}, \mathcal{C}'$ denote two temporal meshes and

$$\mathcal{S} := \{(p, p') \mid p, p' \text{ primitive objects}, p \in \mathcal{C}, p' \in \mathcal{C}'\}$$

the set of pairs of primitive objects in $\mathcal{C}, \mathcal{C}'$. The operation *op* on $\mathcal{C}$ and $\mathcal{C}'$ is called *decomposable*, if there exist operations $\otimes, \oplus$ such that

$$\mathcal{C} \; op \; \mathcal{C}' = \bigotimes_{(p_1,p_2)\in\mathcal{S}} p_1 \oplus p_2$$

Operation $\oplus$ operates on spatio-temporal primitive objects and corresponds to an operation from section 6.2. Operation $\otimes$ combines the results obtained from the $\oplus$-applications. Note that the definition does not require operation $\otimes$ to be computable in $O(1)$.

For example, to detect intersections between two spatio-temporal objects, one sets

$$\begin{aligned} \oplus &\equiv \text{intersects}, \\ \otimes &\equiv \vee \end{aligned}$$

To compute when an intersection occurred between two spatio-temporal objects one sets the operations as follows. Here, $\cup$ denotes the union of temporal elements.

$$\begin{aligned} \oplus &\equiv \text{when\_intersects}, \\ \otimes &\equiv \cup \end{aligned}$$

Hence, the computation of decomposable problems can be described generically, in an iterative manner. This procedure is depicted in algorithm 6.3.1.

6.3.2 Binary Operations without Indexes

In case one of the above binary operations operates upon meshes that have both no indexes, each pair of spatio-temporal primitive objects must be processed in the worst case. However, the algorithm can be described straightforwardly with the above defined concepts. The strategy is to create a *gstepIterator* for the first mesh and to iterate over its primitive objects. For each primitive object, an iterator for the second mesh is created and opened with the

Algorithm 6.3.1 iterative-decomposable-problem(*stObject* o_1, o_2)

```
1: result ⇐ $ne_\otimes$ // neutral element for $\otimes$
2: for each primitive object $p_1 \in o_1$ do
3:     $\mathcal{S} \Leftarrow \{p_2 \mid p_2$ primitive object in $o_2, p_1$.lifetime $\cap\, p_2$.lifetime $\neq \emptyset\}$
4:     for each $p_2 \in \mathcal{S}$ do
5:         result ⇐ result $\otimes (p_1 \oplus p_2)$
6:     end for
7: end for
8: return result
```

temporal interval that equals the lifetime of the current primitive object of the first mesh. While iterating through the second mesh, the corresponding base operation is called upon the current primitive objects, combining the sub-results accordingly. Hence, the procedure resembles a nested-loop processing of a join operation.

6.3.3 Binary Operations with a Single Index

In case one of the above binary operations operates upon meshes, one of which has an index, the operation can be realised in an indexed nested-loop fashion. To this end, for the non-indexed mesh, a *gstepIterator* is created, iterating over the primitive objects. For each primitive object the index of the second mesh is traversed to efficiently localise the relevant primitive objects of the second mesh. Here, relevance depends on the particular operation. For instance, for "intersects" the index-traversal only follows those links to nodes that have intersecting keys. For "distance"-operations, a more detailed description follows in the next section. When a leaf node entry is found to be relevant, a *gstepIterator* is created for its simplex and the temporal interval specified in the entry. Recall that the simplices of a mesh are not necessarily split into primitive objects when indexed. The further processing is that of the nested-loop-like fashion described in the previous section.

6.3.4 Binary Operations with two Indexes

It is common practice to pre-process data in appropriate data structures to be then operated on to solve a given problem efficiently. However, such data structures are not widely available in spatial or spatio-temporal databases, since applying the best data structure for a given problem would entail either to read the whole data set from secondary storage (to build the data structure) or to store and maintain every data structure to be used in persistent memory. Both alternatives are not satisfactory. Index structures for spatio-temporal objects, on the other hand, resulted also from a pre-processing step, though for the main purpose of range queries originally. For these reasons, it seems promising to design algorithms operating on index structures rather than designing specialised data structures for each computational problem.

R-Trees can be used effectively for the computation of intersection- and distance-based operations. This section focuses on how to perform the operation "min_dist" for two temporal meshes. The strategy can be summarised as a synchronised depth-first tree-traversal

Algorithm 6.3.2 recursive-distance(R_1, R_2, $dist$)

```
if (both $R_1$ and $R_2$ are leaves) then
  for all entry-pairs $p$ do
    if $p$.lifetime$_1 \cap p$.lifetime$_2 \neq \emptyset$ then
      $v \Leftarrow$ compute min_dist using no-index-algorithm (section 6.3.2)
      $dist \Leftarrow \min\{dist, v\}$;
    end if
  end for
  return $dist$;
end if
$pairs \Leftarrow \emptyset$
for all $e_1 \in R_1$.root.entries do
  for all $e_2 \in R_2$.root.entries do
    if $e_1$.lifetime $\cap$ $e_2$.lifetime $\neq \emptyset$ then
      insert $p = (e_1, e_2)$ into $pairs$
      $p.minDist \Leftarrow$ compute minimum distance between MBBs
      $p.maxDist \Leftarrow$ compute maximum distance between MBBs {definition see text}
      $dist \Leftarrow \min\{dist, p.maxDist\}$
    end if
  end for
end for
sort $pairs$ on $p.minDist$, smallest first
$p \Leftarrow$ pop first element from $pairs$
while any pairs to process do
  $v \Leftarrow$ recursive-distance($p.subtree_1$,
                                   $p.subtree_2, dist$)
  $dist \Leftarrow \min\{dist, v\}$
  $p \Leftarrow$ pop next element from $pairs$
  if $p.minDist \geq dist$ then
    return $dist$
  end if
end while
return $dist$
```

algorithm.

The algorithm *recursive-distance* (algorithm 6.3.2) is called by the auxiliary statement:
`recursive-distance(`R_1`.root, `R_2`.root, `∞`);`
Thus, algorithm *recursive-distance* receives as input two R-trees as well as the distance so far computed, initialized to infinity for the first call on the roots of the trees. The key idea is to draw conclusions about which subtree-pairs may be pruned and which to follow next in the depth-first search and to update the current distance value on each level of recursion. Subtree-pairs, whose temporal interval of validity does not intersect, need not be considered. Subtree-pairs, whose objects have a distance guaranteed to be bigger than the current distance-value, can be neglected as well. More precisely, if the current directory MBBs (the MBBs of an inner R-tree node) have n and m entries, respectively, the algorithm expands all $n \times m$ directory MBB-pairs in main memory and computes the minimum distance and the maximum distance between each MBB-pair (lines 11–20). It thereby prunes those pairs the lifetime of which does not intersect (line 13). The maximum distance between two MBBs b_1, b_2 is defined as

$$\max\{d(p_1, p_2) \mid p_1 \in b_1, p_2 \in b_2\}$$

The rationale behind the minDist and maxDist of an MBB-pair is that the distance of the *objects*, which the boxes approximate, is guaranteed to be within the interval [minDist,maxDist]. Hence, the current level of recursion is allowed to adapt the current *dist*-value to the smallest maxDist-value (line 17). Note that the well-known minMax-dist between MBBs b_1, b_2, defined as

$$\min\{d_{max}(f_1, f_2) \mid f_i \text{ is a face of } b_i\} \text{ and } d_{max}(f_1, f_2) = \max\{d(p_1, p_2) \mid p_i \in f_i\}$$

cannot be used here, as the MBBs approximate the *trajectories* of spatio-temporal objects such that the minMax-dist holds only for the trajectories, whereas the objects may have been at the specific locations at different times. After processing each MBB-pair, the algorithm performs then a quicksort (line 21) on the list of pairs where the sorting key is the distance between the MBB-pair. The first element of this list of pairs is extracted and the corresponding subtrees are then searched. When the recursive procedure returns to the current level with an updated distance-value v (line 24), the next subtrees are only searched if their MBB minDist is smaller than v (line 27). If, on the other hand, v is less or equal, the remaining subtree-pairs can be pruned (note the ordering of the *pairs*-list).

6.3.1. PROPOSITION. *Algorithm 6.3.2 terminates and reports the minimum Euclidean distance $d(o_1, o_2)$ of its input spatio-temporal objects.*

Proof: Computing the distances of all possible primitive simplex pairs $\{(s_1, s_2) \mid s_1 \in o_1, s_2 \in o_2\}$ and returning their minimum obviously yields the result. This is what algorithm 6.3.2 does (line 4 for leaves and line 15 for inner nodes of the R-trees) except for subtree pruning. A subtree pair is only pruned if its *p.minDist* is not less than the current *dist* (line 13). This pruning is justified. It remains to show that at each stage of the algorithm $dist \geq d(o_1, o_2)$ holds true. Four cases occur: (1) during initialisation it holds true, since $dist = \infty$; (2) in line 5 it holds true, since distances between simplices are computed; (3) in line 17 it holds true; (4) in line 25 it holds true. □

The presented algorithm has been targeted at the computation of the minimum Euclidean distance between two meshes. The adaptation to the further operations mentioned above can be easily accomplished.

Chapter 7

Conclusions

This thesis aims at developing methods for representing, storing, and retrieving spatio-temporal geo-objects within a database system. Thereby, temporal change concerns both thematic and spatial attributes of geo-objects; however, while temporal change of thematic attributes falls into the domain of temporal database systems (Snodgrass, 1992), temporal change of spatial data is still a novel research domain of database systems. Therefore, this thesis focuses on the management of changing spatial object.

At present, applications that must handle such time-dependent objects are not well supported by state-of-the-art GIS- and database tools (Peuquet, 2001) and would thus benefit from integrated concepts for their management. Obviously, the concepts must go beyond the storage of snapshots of objects at time-steps. Main reasons are the redundancy of this storage scheme (constant parts of a changing geometry are repeated at every snapshot), as well as insufficient query facilities. Applications, the geometries of which are subject to *continuous* change, emphasise the latter aspect, since query conditions about object states between two snapshots cannot be formulated without further ado (Yeh and de Cambray, 1995).

In particular, the following concepts were the centre of interest of this thesis. First, it was investigated how to represent spatio-temporal objects within a database system. The research was conducted by formalising and extending a model that has been introduced within the collaborative research centre 350, located at Bonn University (Neugebauer and Simmer, 2003). The model facilitates the representation of geometries that undergo both discrete and continuous change. Secondly, a type system has been derived out of the representational model that enables the instantiation of spatio-temporal objects on the logical level of a database system. Hence, this data is treated as a first-class citizen. Third, concepts for the internal level have been researched that form a realisation of the type system. Fourth, an inventory of operations for these objects on the internal level has been derived.

Sellis (1999) delineated the research to be undertaken in the field of spatio-temporal database management systems (STDBMSs) and mentions the following aspects in particular: (1) 'devising data models and operators with clean and complete semantics'; (2) 'efficient implementations of these models and operators'; (3) 'work on indexing and query optimisation'; and (4) 'experimentation with alternative architectures for building STDBMSs'. The research of this thesis integrates into these aspects as follows. The representational model is a contribution to (1); thereby, the model and its operations have been described formally. The realisation on the internal level subsumes under (2), while work pertaining to (3) is limited to an extension of the spatial index structure R* tree (Beckmann et al., 1990) and also query optimisation has been excluded. Aspect (4) applies insofar as the realisation environment rests on ObjectStore$^{\text{TM}}$, a commercial object storage system.

Worboys (1994) presents a representational model that is based on simplicial complexes and allows for change to the geometry of objects in discrete steps. This model is cited frequently. For instance, Chomicki and Revesz (1999a) extend the model such that also continuous change of an object's geometry becomes representable. Thus, this model is related to the one proposed in this thesis, although the former does not allow a separation of vertices from meshes and does not map the model on separate data types other than relations and tuples. A system of abstract data types for the representation of 2D objects, which change location and shape continuously, has been introduced by Güting et al. (2000). First steps towards an implementation of this system are described by Forlizzi et al. (2000). Recent surveys of further representational models for spatio-temporal data have been given by Peuquet (2001) and Abraham and Roddick (1999). The book by Ott and Swiaczny (2001) offers a survey from a more practical perspective, focusing on discrete change to spatial data. The book by Koubarakis et al. (2003) gives an overview of the research conducted by the Chorochronos project. On the whole, it must be observed that—with the exception of the field of index structures—few work has been targeted at internal data structures and operations for spatio-temporal database systems.

The presentation of the results of this thesis is oriented towards the above mentioned classification and is structured as follows: (1) representational model and class hierarchy for spatio-temporal objects; (2) concepts for the internal realisation of the class hierarchy; (3) operations on the objects and their realisation.

The representational model for spatio-temporal objects has its roots in experiences gained during a collaborative research centre hosted at Bonn University (Neugebauer and Simmer, 2003). The contributions added in this thesis are the consolidation of the main aspects as seen from a database system's perspective (see also Shumilov and Siebeck (2001) for a preliminary presentation and chapter 3 in this thesis). The work results in a novel, formalised representational model. The basic idea that the location of vertices of triangular meshes are functions of time has been extended by time-stamping the elements of a simplicial complex with a lifespan and by integrating certain consistency constraints on the data. The concepts "temporal simplex", "temporal complex", and "temporal mesh" have been defined and make the extensions precise. Treating spatio-temporal objects as functions from time into space, a spatio-temporal database system must also account for temporal types (marking off the domain of such functions). Therefore, requirements for temporal types have been investigated that were also included into the model. Treating spatio-temporal objects as time-dependent point sets, it makes sense to investigate the closure of set-oriented operations like "intersection" or "union" on such point sets in their representation. An important result is the non-closure of these and further operations, even if applied to a spatio-temporal object and an (atemporal) spatial object.

The representational model has been transferred into a class hierarchy on the user level.

The design was guided by the following aspects. First, the defined concepts of a temporal simplex and a temporal mesh are supported directly by introducing a separate class for each. Second, to support interactive geometric modelling, the classes contain update operations, i.e. the resulting data types are dynamic. Third, time-dependent vertices are separated from the meshes in which they occur; to this end, mesh elements hold references to their incident vertices. Fourth, the object-oriented concept of substitution polymorphism (Abadi and Cardelli, 1996) should retain its semantic. This concept facilitates in particular the assignment of user-defined, thematic attributes to the elements of a mesh through inheritance. Though supported by the underlying object storage system, it is thwarted by a further design goal: to separate thematic from geometric data for the purpose of grouping on secondary storage. Therefore, the facility of substitution polymorphism had to be assured by a special mechanism. Finally, the integration of spatio-temporal classes was carried out *transparently*, that means (atemporal) spatial objects are regarded as a special case of spatio-temporal objects and are represented by the same classes.

On the internal level, thematic attributes are separated from geometric attributes. Nevertheless, a connection between both parts of a time-dependent geo-object exists such that the thematic part is accessible from the geometric part and vice-versa. This problem could be solved by application of the design pattern "bridge" (Gamma et al., 1995). The positive effect is twofold. First, thematic attributes and geometric attributes can be grouped on secondary storage independent from each other. Second, the geometric part can be coupled with different internal representations; this way, spatio-temporal objects, the geometry of which does not change with time (atemporal objects), can be managed by proven techniques of spatial database systems (e.g. indexes, algorithms, etc.) and the problem of transparent integration (see above) can thus be solved practically. Grouping of the geometric part of a spatio-temporal objects on secondary storage was realised by a clustering index. To this end, the R* tree spatial index (Beckmann et al., 1990) has been adapted accordingly.

Besides these architecture-driven concepts, the realisation of operations has been investigated. Among these operations are updates, basic retrieval like generating a snapshot of a spatio-temporal object for a given time-step, spatio-temporal range queries, an intersection-predicate, and functions for computing the minimum Euclidean distance between spatio-temporal objects. Update operations include also the management of neighbour-relationships between temporal simplices of a temporal mesh, maintaining connected components of a temporal mesh, as well as checking the consistency constraints on temporal simplices and meshes.

Beyond the presented and related work, many research questions remain open to be investigated. In the following, some selected issues are addressed. Strongly tied to this thesis is the question of placement of object headers (see section 5.6.1). Currently, allocating spatio-temporal object headers ignores spatio-temporal information. Especially when dealing with very large data sets, it could be interesting to investigate the approach of an 4D R-Tree-driven allocation. In this scenario, a page of object headers forms the value of an R-Tree leaf entry. On insertion of a new object header, the R-Tree "ChooseLeaf"-operation (Guttman, 1984) determines the placement of the header. The approach aims at reducing I/O during retrieval operations, similar to the BFS-based explorations of temporal meshes.

A further research question lies in how to implement operators that support a qualitative spatio-temporal reasoning, e.g., in the line of Galton (1995); Muller (1998); Erwig and

Schneider (1999); Hazarika and Cohn (2001); Erwig and Schneider (2002). While in the spatial setting qualitative topological relationships between spatial objects have found their way into query languages, the implementation for the spatio-temporal setting remains a challenge. Here, it would be interesting, if and how such operators can be implemented within the introduced representational model.

Instead of extending the set of operations on the spatio-temporal objects, another line of research is their robustness in the presence of floating point arithmetic. As was shown in chapter 6, even a basic operation involves a huge number of floating point operations, a potential source of imprecision (Goldberg, 1991). An interesting approach could be the application of the adaptive floating point arithmetic of Shewchuk (1997).

Yet a different direction can be taken by extending the representational model for spatio-temporal objects. For instance, the model of point movement can be more powerful, which was here limited to be piecewise linear. Although the linear approximations seemed appropriate for the projects presented in section 3.1, approximations, e.g., based on splines could be more accurate and/or space-efficient. Extending the model of point movement invalidates many of the presented operations, as they expect linear movement.

Finally there is the goal of a consensus representational model for spatio-temporal data. Such a model would parallelise the spatial OGC model and would foster the development of methods and tools for storage, retrieval, visualisation, analysis, and exchange of data; at present, a visionary goal.

Bibliography

M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

T. Abraham and J. F. Roddick. Survey of spatio-temporal databases. *Geoinformatica*, 3(1): 61–99, 1999.

S. Abramowski and H. Müller. *Geometrisches Modellieren*. BI Wissenschaftsverlag, 1991.

P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. of the 19th Symposium on Principles of Database Systems*, pages 175–186, 2000.

J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

R. Alms, O. Balovnev, M. Breunig, A. B. Cremers, T. Jentzsch, and A. Siehl. Space-time modelling of the lower Rhine basin supported by an object-oriented database. *Physics and Chemistry of the Earth*, 23(3):251–260, 1998.

O. Balovnev, T. Bode, M. Breunig, A. B. Cremers, W. Müller, G. Pogodaev, S. Shumilov, J. Siebeck, A. Siehl, and A. Thomsen. The story of the GeoToolKit – an object-oriented geodatabase kernel system. *GeoInformatica*, 2003. Accepted for publication.

O. Balovnev and M. Breunig. Erste Versuche zur Modellierung der Zeit als Anwendung eines GeoToolKits. In *Zeit als weitere Dimension in Geo-Informationssystemen*, pages 107–114. University of Rostock, Institute of Geodesy and Geoinformatics, Technical Report No. 7, 1997.

O. Balovnev, M. Breunig, and A. B. Cremers. *From GeoStore to GeoToolKit: The Second Step*, volume 1262 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 1997.

O. Balovnev, M. Breunig, A. B. Cremers, and M. Pant. Building geo-scientific applications on top of geotoolkit: a case study of data integration. In *Proc. of the 10th International Conference on Scientific and Statistical Database Management*, pages 260–269, 1998.

O. Balovnev, M. Breunig, A. B. Cremers, and S. Shumilov. Space-time modelling of the lower Rhine basin supported by an object-oriented database. In M. Goodchild, M. Egenhofer, R. Fegeas, and C. Kottmann, editors, *Interoperating Geographic Information Systems.* Kluwer Academic Publishers, 1999.

R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal: Very Large Data Bases*, 5(4):264–275, December 1996. Electronic edition.

L. Becker, L. Bernard, H. Ditt, K.H. Hinrichs, B. Schmidt, U. Streit, and A. Voigtmann. Ansätze zur Integration eines dynamischen Atmosphärenmodells in einen objektorientierten GIS-Kern. In *Zeit als weitere Dimension in Geo-Informationssystemen*, pages 97–105. Universität Rostock, 1997.

N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 322–331, 1990.

J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

L. Bernard, B. Schmidt, U. Streit, and C. Uhlenküken. Managing, modeling, and visualizing high-dimensional spatio-temporal data in an integrated system. *GeoInformatica*, 2(1):59–77, 1998.

E. Bertino, E. Ferrari, G. Guerrini, and I. Merlo. Extending the ODMG object model with time. In *Proc. of the 12th European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 41–65. Springer, 1998.

A. Biliris and E. Panagos. A high performance configurable storage manager. In *Proc. of the 11th IEEE International Conference on Data Engineering*, pages 35–43, 1995.

M.H. Böhlen, C.S. Jensen, and B. Skjellaug. Spatio-temporal database support for legacy applications. In *ACM Sympos. on Applied Computing*, pages 226–234, 1998.

M. Breunig. *Integration of Spatial Information for Geo-Information Systems*, volume 61 of *Lecture Notes in Earth Sciences, 171 pages.* Springer, Heidelberg, 1996.

M. Breunig. *On the Way to Component-Based* 3D/4D *Geo-Information Systems*, volume 94 of *Lecture Notes in Earth Science, 199 pages.* Springer, Heidelberg, Germany, 2001.

M. Breunig, A. B. Cremers, W. Müller, and J. Siebeck. New methods for topological clustering and spatial access in object-oriented 3D databases. In *Proc. of the 9th ACM International Symposium on Advances in Geographic Information Systems*, 2001.

M. Breunig, A. B. Cremers, W. Müller, and J. Siebeck. Examination of database supported spatio-temporal intersection queries. In *Proc. of the 5th AGILE Conference on Geographic Information Science*, 2002.

T. Brinkhoff. Requirements of traffic telematics to spatial databases. In *Proc. of the 6th International Symposium on Large Spatial Databases*, number 1651 in Lecture Notes in Computer Science, pages 365–369. Springer, 1999.

M. Cai, D. Keshwani, and P. Z. Revesz. Parametric rectangles: A model for querying and animation of spatiotemporal databases. In *Proc. of the Conference on Extending Database Technology*, volume 1777 of *Lecture Notes in Computer Science*, pages 430–444. Springer, 2000.

C. Xinmin Chen and C. Zaniolo. SQL ST : A spatio-temporal data model and query language. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 96–111, 2000.

J. Chomicki and P. Z. Revesz. Constraint-based interoperability of spatiotemporal databases. *GeoInformatica*, 3(3):211–243, 1999a.

J. Chomicki and P. Z. Revesz. A geometric framework for specifying spatiotemporal objects. In *Proc. of the 6th Intl. Workshop on Temporal Representation and Reasoning*, 1999b.

C. Claramunt, C. Parent, and M. Theriault. Design patterns for spatio-temporal processes. In S. Spaccapietra and F. Maryanski, editors, *Searching for Semantics: Data Mining, Reverse Engineering*, pages 415–428. Chapman & Hall, 1997.

D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* The MIT Press and McGraw-Hill Book Company, 1989.

S. Dieker and R. H. Güting. Plug and play with query algebras: SECONDO-a generic DBMS development environment. In *International Database Engineering and Application Symposium*, pages 380–392, 2000.

J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

M. J. Egenhofer, A. U. Frank, and J. P. Jackson. A topological data model for spatial databases. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, editors, *Proc. of the 1st Symposium SSD on Design and Implementation of Large Spatial Databases*, volume 409 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 1990.

M. J. Egenhofer, J. Glasgow, O. Günther, J. R. Herring, and D. J. Peuquet. Progress in computational methods for representing geographical concepts. *International Journal of Geographical Information Science*, 13(8):775–796, 1999.

M. Erwig and M. Schneider. Developments in spatio-temporal query languages. In *Proc. of the IEEE Int. Workshop on Spatio-Temporal Data Models and Languages*, pages 441–449, 1999.

M. Erwig and M. Schneider. Spatio-temporal predicates. *IEEE Transactions on Knowledge and Data Engineering*, 2002.

ESRI. ArcStorm – the ARC storage manager. ESRI white paper, Environmental Systems Research Institute, Inc., 1996.

ESRI. ArcView tracking analyst extension - the solution for temporal analysis. ESRI white paper, Environmental Systems Research Institute, Inc., 1998.

L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 319–330. ACM Press, 2000.

S. K. Gadia and C.-S. Yeung. A generalized model for a relational temporal database. In *Proc. of the Intl. ACM Conference on Management of Data*, pages 251–259, 1988.

A. Galton. Towards a qualitative theory of movement. In *Proc. of the 2nd International Conference on Spatial Information Theory*, number 988 in Lecture Notes in Computer Science, pages 377–396. Springer, 1995.

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

D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

S. Gottschalk. Collision queries using oriented bounding boxes, 1998.

T. Griffiths, A. A. A. Fernandes, N. Djafri, and N. W. Paton. A query calculus for spatio-temporal object databases. In *Proc. of the 8th International Symposium on Temporal Representation and Reasoning*, pages 101–110, 2001a.

T. Griffiths, A. A. A. Fernandes, and N. W. Paton. An ODMG-compliant spatio-temporal data model. Technical Report CSPP-14, Department of Computer Science, University of Manchester, 2001b.

T. Griffiths, A. A. A. Fernandes, N. W. Paton, K. Mason, B. Huang, M. Worboys, C. Johnson, and J. Stell. Tripod: A comprehensive system for the management of spatial and aspatial historical objects. In *Proc. of the 9th ACM Int. Symposium on Advances in Geographic Information Systems*, pages 118–123, 2001c.

T. Griffiths, A.A.A. Fernandes, N.W. Paton, and N. Djafri. Realm-based temporal data types. Preprint CSPP-15, Department of Computer Science, University of Manchester, 2002.

S. Grumbach, P. Rigaux, and L. Segoufin. Spatio-Temporal Data Handling with Constraints. In *ACM International Workshop on Advances in Geographic Information Systems (ACM-GIS)*, pages 106–111. ACM Press, 1998.

R. H. Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–400, 1994.

R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems*, 25(1):1–42, 2000.

R. H. Güting and M. Schneider. Realm-based spatial data types: The ROSE algebra. *VLDB Journal: Very Large Data Bases*, 4(2):243–286, 1995.

A. Guttman. *R*-trees: a dynamic index structure for spatial searching. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 14(2):47–57, 1984.

M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proc. of the 8th Conference on Extending Database Technology*, pages 251–268, 2002.

T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

S. M. Hazarika and A. G. Cohn. Qualitative spatio-temporal continuity. In *Proc. of COSIT*, pages 92–107, 2001.

C. S. Jensen, C. E. Dyreson, M. Boehlen, and J. Clifford. *The Consensus Glossary of Temporal Database Concepts — February 1998 Version*, volume 1399 of *Lecture Notes in Computer Science*, pages 367–405. Springer, 1998.

D. Jungnickel. *Graphen, Netzwerke und Algorithmen.* BI Wissenschaftsverlag, 1990.

D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *International Workshop on Spatio-Temporal Database Management*, pages 119–134, 1999.

G. Kollios, V. J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *Knowledge and Data Engineering*, 13(5):758–777, 2001.

M. Koubarakis, T. Sellis, A. Frank, S. Grumbach, R. H. Güting, C. Jensen, N. Lorentzos, Y. Manolopoulos, E. Nardelli, B. Pernici, H.-J. Schek, M. Scholl, B. Theodoulidis, and N. Tryfona, editors. *Spatiotemporal Databases: The Chorochronos Approach*, volume 2520 of *Lecture Notes in Computer Science.* Springer, 2003.

G. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint databases.* Springer, 2000.

C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

P. Lienhardt. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal of Computational Geometry and Applications*, 4(3):275–324, 1994.

J. Ma and B. Knight. A general temporal theory. *Computer Journal*, 37(2):114–123, 1994.

J.-L. Mallet. GOCAD: A computer aided design program for geological applications. In A. K. Turner, editor, *Three-Dimensional Modeling with Geoscientific Information Systems*, pages 123–142. Kluwer Academic Publishers, 1992.

J.-L. Mallet. *Geomodeling.* Oxford University Press, 2002.

M. Mäntylä. *An Introduction to Solid Modeling.* Computer Science Press, Rockville, MD, 1988.

K. Mehlhorn. *Data Structures and Algorithms: 3. Multidimensional Searching and Computational Geometry.* Springer, Berlin, 1984.

Philippe Muller. A Qualitative Theory of Motion Based on Spatio-Temporal Primitives. In *Proc. of the Principles of Knowledge Representation and Reasoning Conference (KR'98)*, pages 179–187. Morgan Kaufmann, 1998.

M. Nascimento and J. Silva. Towards historical r-trees. In *Proc. of the ACM Symposium on Applied Computing*, pages 235–240, 1998.

M.A. Nascimento, R.O. Silva, and Y. Theodoridis. Evaluation of access structures for discretely moving points. In *Proc. of the International Workshop on Spatio-Temporal Database Management*, volume 1678 of *Lecture Notes in Computer Science.* Springer, 1999.

H. J. Neugebauer and C. Simmer, editors. *Dynamics of Multiscale Earth Systems*, volume 97 of *Lecture Notes in Earth Sciences.* Springer, 2003.

ObjectStore. *Advanced C++ API User Guide.* Progress Software Corp., 2003.

T. Ott and F. Swiaczny. *Time-Integrative Geographic Information Systems.* Springer, 2001.

D. Papadopoulos, G. Kollios, D. Gunopulos, and V. J. Tsotras. Indexing mobile objects using duality transforms. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2002.

C. Parent, S. Spaccapietra, and E. Zimányi. Spatio-temporal conceptual models: Data structures + space + time. In C. Bauzer Medeiros, editor, *Proc. of the 7th International Symposium on Advances in Geographic Information Systems*, pages 26–33. ACM, 1999.

A. Di Pasquale, L. Forlizzi, C. S. Jensen, Y. Manolopoulos, E. Nardelli, D. Pfoser, G. Proietti, S. Saltenis, Y. Theodoridis, T. Tzouramanis, and M. Vassilakopoulos. Access methods and query processing techniques. In M. Koubarakis, T. Sellis, A. Frank, S. Grumbach, R. H. Güting, C. Jensen, N. Lorentzos, Y. Manolopoulos, E. Nardelli, B. Pernici, H.-J. Schek, M. Scholl, B. Theodoulidis, and N. Tryfona, editors, *Spatio-Temporal Databases: The Chorochronos Approach*, volume 2520 of *Lecture Notes in Computer Science.* Springer, 2003.

D. J. Peuquet. Making space for time: Issues in space-time data representation. *GeoInformatica*, 5(1):11–32, 2001.

D. J. Peuquet and L. Qian. An integrated database design for temporal gis. In *Proc. of the 7th International Symposium on Spatial Data Handling*, 1996.

D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proc. of 26th International Conference on Very Large Data Bases*, pages 395–406. Morgan Kaufmann Publishers, 2000.

K. Polthier and M. Rumpf. A concept for time-dependent processes. In M. Göbel, H. Müller, and B. Urban, editors, *Visualization in Scientific Computing*, pages 137–153. Springer, 1994.

C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. Star-tree: An efficient self-adjusting index for moving points. In *Proc. of the 4th Workshop on Algorithm Engineering and Experiments*, 2002.

A. Raza and W. Kainz. Cell tuple based spatio-temporal data model: An object oriented approach. In Claudia Bauzer Medeiros, editor, *Proc. of the 7th International ACM Symposium on Advances in Geographic Information Systems*, pages 20–25. ACM, 1999.

P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases With Applications to GIS.* Morgan Kaufmann Publishers, 2002.

D. Rouby, Th. Souriot, J.P. Brun, and P.R. Cobbold. Displacement, strains, and rotations within the Afar depression (Djibouti) from restoration in mapview. *Tectonics*, 15(5):952–965, 1996.

S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD Conference*, pages 331–342, 2000.

T. Sellis. Research issues in spatio-temporal database systems. In R.H. Güting, D. Papadias, and F. Lochovsky, editors, *Advances in Spatial Databases*, volume 1651 of *Lecture Notes in Computer Science*. Springer, 1999.

J.R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom.*, 18, 1997.

S. Shumilov and J. Siebeck. Database support for temporal 3D data: Extending the GeoToolKit. In *Proc. of the 7th EC-GI and GIS Workshop*, Potsdam, Germany, 2001.

A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Quering the uncertain position of moving objects. In *Temporal Databases – Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*, pages 310–337. Springer, 1998.

P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proc. 13th Int. Conf. on Data Engineering*, 1997.

R. Snodgrass. Temporal databases. In *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, number 639 in Lecture Notes in Computer Science, pages 22–64. Springer, 1992.

R. T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Kafer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. A. Sripada. TSQL2 language specification. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(1):65–86, 1994.

S. Spaccapietra, C. Parent, and E. Zimanyi. Modeling time from a conceptual perspective. In G. Gardarin, J. French, N. Pissinou, K. Makki, and L. Bouganim, editors, *Proc. of the ACM*

International Conference on Information and Knowledge Management, pages 432–440, New York, 1998. ACM Press.

A. U. Tansel, J. Clifford, S. Gadia, S. Jajopia, A. Segev, and D. R. Snodgrass, editors. *Temporal Databases.* Series on Database Systems and Applications. Benjamin/Cummings Publishing, Redwood City (CA), USA, 1993.

Y. Tao and D. Papadias. The MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Proc. of the 27th VLDB Conference*, 2001.

R. E. Tarjan. *Data structures and network algorithms*, volume 44 of *CBMS-NSF Reg. Conf. Ser. Appl. Math.* SIAM, 1983.

J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998. ISSN 0010-4620.

Y. Theodoridis, T. Sellis, A.N. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *10th International Conference on Scientific and Statistical Database Management.* IEEE Computer Society, 1998.

Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-temporal indexing for large multimedia applications. In *International Conference on Multimedia Computing and Systems*, pages 441–448, 1996.

A. Thomsen and A. Siehl. Towards a balanced 3D kinematic model of a faulted domain – the Bergheim open pit mine. *Geol. Mijnbouw*, 2000.

T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees and spatio-temporal query processing. *The Computer Journal*, 43(4):325–324, 2000.

M. Van Kreveld. Digital elevation models and TIN algorithms. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Wiedmayer, editors, *Algorithmic foundations of geographic information systems*, number 1340 in Lecture Notes in Computer Science, chapter 3, pages 37–78. Springer, 1997.

A. Voigtmann, L. Becker, and K. H. Hinrichs. Temporal extensions for an object-oriented geo-data-model. In *Proc. of the 7th International Symposium on Spatial Data Handling*, pages 25–41, Delft, Netherlands, 1996.

D. Watt. *Programming Languages - Concepts and Paradigms.* Prentice Hall International, 1990.

K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *Computer Graphics and Applications*, 5(1):21–40, 1985.

K. Weiler. The radial edge structure: a topological representation for non-manifold geometric boundary modeling. In *Proc. of the IFIG WG 5.2*, 1986.

S. J. White and D. J. DeWitt. QuickStore: A high performance mapped object store. *VLDB Journal: Very Large Data Bases*, 4(4):629–673, 1995.

O. Wolfson. Moving objects information management: The database challenge. In *Proc. of the 5th International Workshop on Next Generation Information Technologies and Systems*, number 2382 in Lecture Notes in Computer Science, pages 75–89. Springer, 2002.

O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proc. of the 10th International Conference on Scientific and Statistical Database Management*, pages 111–122, 1998.

M. F. Worboys. *GIS: A computing perspective.* Taylor & Francis, 1995.

Michael F. Worboys. A Unified Model for Spatial and Temporal Information. *The Computer Journal*, 37(1):26–34, 1994.

X. Xu, J. Han, and W. Lu. Rt-tree: An improved r-tree index structure for spatiotemporal databases. In *Proc. of the 4th International Symposium on Spatial Data Handling*, 1990.

T. S. Yeh and B. de Cambray. Modeling highly variable spatio-temporal data. In *Proc. of the 6th AustraliAsian Database Conference*, pages 221–230, 1995.

T. S. Yeh and P. Feautrier. A unified spatial and spatiotemporal model using polyhedra. Technical report, Versailles University, 1998.

W. Zhang and G. J. Hunter. Temporal interpolation of spatially dynamic objects. *GeoInformatica*, 4(4):403–418, 2000.

Abstract

The quickly increasing number of spatio-temporal applications in fields like environmental management or geology is a new challenge to the development of database systems. This thesis addresses three areas of the problem of integrating spatio-temporal objects into databases. First, a new representational model for continuously changing, spatial 3D objects is introduced and transferred into a small system of classes within an object-oriented database framework. The model extends simplicial cell complexes to the spatio-temporal setting. The problem of closure under certain operations is investigated. Second, internal data structures are introduced that represent instances of the (user-level) spatio-temporal classes. A new technique provides a compromise between compact storage and efficient retrieval of spatio-temporal objects. These structures correspond to temporal graphs and support updates as well as the maintainance of connected components over time. Third, it is shown how to realise further operations on the new type of objects. Among these operations are range queries, intersection tests, and the Euclidean distance function.