Representing Topological Relationships for Moving Objects

Erlend Tøssebro[1] and Mads Nygård[2]

[1] Department of Electronics and Computer Science, University of Stavanger, NO-4036 Stavanger, Norway
erlend.tossebro@uis.no
[2] Department of Computer and Information Science, Norwegian University of Science and Technology, NO-7491 Norway
mads@idi.ntnu.no

Abstract. Several representations have been created to store topological information in normal spatial databases. However, not that much work has been done to store such relationships for spatiotemporal data. This paper extends the representation of moving objects from [7] so that it can also store and enforce some of the topological relationships between the objects. This is done in a fashion similar to the Node-Arc-Area model for normal spatial databases. One use of such a representation is storing a changing spatial partition.

1 Introduction

For purely spatial databases, there are several ways to represent topology. One such way is the Node-Arc-Area representation [13]. Each line segment stores a link to the two regions that it borders, each point stores a link to each line segment that begins or ends in that point and each face (connected region) stores a link to at least one of its border curves. This ensures that if the border of one region is updated, the borders of all other regions are automatically updated as necessary to maintain known topological relationships.

An important use for topological information is storing a spatial partition, since without topological information it is difficult to control whether a given set of regions forms a partition or not. [5] presents an abstract model for spatiotemporal partitions called the "honeycomb model". This is called an abstract model in this paper because it is based on infinite point sets. A discrete model, by contrast, is based on constructs that one could realistically store in a database such as straight line segments. Both raster and vector models are discrete models. The "honeycomb" model represents time as an extra dimension and represents a spatiotemporal partition as a three-dimensional partition with the limitation that at any time instant the partition should be a legal two-dimensional partition.

However, no discrete model for spatiotemporal partitions is known to the authors. Nor is a discrete model of spatiotemporal information that takes topological relationships into account.

M. Raubal et al. (Eds.): GIScience 2006, LNCS 4197, pp. 383 – 399, 2006.

1.1 Examples of Moving Partitions

Here is a collection of examples of where the ability to store a spatiotemporal partition might be useful. The first was also mentioned in [5].

Example 1: Subdivision of the world into countries: The countries of the world make up a partition because they cover all the land and do not overlap each other. The borders of countries may also change (such as when east and west Germany merged or when Yugoslavia split apart), but only in discrete steps.

Example 2: Land cover: The type of vegetation that covers different areas changes continuously over time. It would be theoretically possible to monitor these changes very often, but the mapping agencies do not have the manpower for this. So instead snapshots are created that may be decades apart. To visualize the changes in land cover it is better to produce an interpolation that yields a best guess as to how the borders moved than just do discrete jumps. Land cover within a given area might be updated simultaneously. However, land cover regions neighbouring this area may be updated at entirely different times.

Example 3: Soil type classification: All land regions have a soil type and the classes are usually distinct, thus making this into a partition. Soil type may also change continuously in time. Usually this change is very slow, measured over millennia or more, but in some cases changes may happen far more rapidly, such as when forest cover is removed and erosion becomes much higher than it was.

1.2 Examples of Topology Not Involving a Partition

Here are some examples of situations in which storing explicit topology might be useful even though they do not involve partitions:

Example 4: Visualization of how the landscape has looked in the past. Landscapes change. Rivers alter their courses and lakes grow smaller or larger over time. Glaciers grow and shrink. In this case one not only needs to create temporal version of the objects, but one also needs to "glue" them together to avoid noticeable inconsistencies. For instance, if a river flows out of a lake, it should end at the lake rather than just next to it or slightly inside it.

Example 5: In an ordinary map database, one might have a glacier that ends over a lake. Both the glacier and the lake may grow or shrink over time, but they often border each other.

Example 6: When a major oil spill occurs, one might want to find out if any animals for which its position is known were inside the spill area at any time.

2 Related Work

Representations for topological relationships have been studied extensively for purely spatial databases as well as for spatiotemporal databases with step wise discrete changes. This section describes those earlier works that this paper directly builds on.

[10] describes a temporal topology that he calls "tropology". This tropology describes the possible chains of events that may occur for a single object. It does not,

however, deal with multiple connected objects and does not assume any particular storage model.

A system for reasoning about changes in the topological relationships between moving objects is described in [3].

[7] describes a discrete model for independent spatiotemporal objects. This model is based on time slices. A time slice is a period of time in which the object moves according to a simple function. Points move linearly. The end points of line segments can move like other points except that the line segment is not allowed to rotate. Area objects are represented by their border lines. A rotating line segment is represented as two line segments that shrink to points in one or the other end of the time slice.

[12] describes a method for generating the representation of faces (area objects that consist of only one connected component) from [7] using snapshots of the faces.

For pure spatial data several vector representations have been created that represent topology explicitly. The Node-Arc-Area (NAA) representation that is presented in for instance [13] is one of them. In the NAA representation, lines store the area objects that are to the left and right of the line as well as their start and end points. The border of an area object is thus defined in the line objects. Thus if the border of an area is updated, the borders of its neighbours are also automatically updated.

[11] describes how to implement topological relationships on complex regions using plane-sweep algorithms on a realm. That paper says nothing on how to make that realm accurate from the outset.

[6] formally describes how to handle topological predicates in spatiotemporal database systems. Their basic method is to *lift* spatial predicates to the spatiotemporal case. Lifting a predicate converts a spatial predicate into a spatiotemporal one with a temporally varying output. For any time instant the output of the lifted predicate is the same as for the spatial predicate with the same objects at that time instant.

They then define quantification on these such that one might ask whether the predicate is true at some point in the time interval or over the whole time interval. They then use the universally quantified operations to define temporal aggregations, for example *Enters*. A point p *Enters* a region R if p starts outside the region, then meets it and then is inside it. They further show that such lifting and defining temporal predicates gives a far more expressive language than using the basic Egenhofer relations from [4] on 3D objects.

[2] defines spatiotemporal objects by an initial snapshot and a transformation function. Some closure properties of this model are then analysed. For instance, for a linear transformation function, the model is closed under union, intersection and difference for rectangles, but only under union for arbitrary polygons.

[1] and [8] define discrete models for spatiotemporal data based on constraints. A convex region is defined as a set of linear constraints (such as: $x + 2y \leq 5$). A non-convex region is defined as a union of convex regions. The advantage of such a system is that it can be easily extended to arbitrary dimensions and can use ordinary relational algebra operations to express many spatial operations that need special operators in abstract data type approaches.

One disadvantage of this approach is that it does not take topology into consideration. A region is stored as a set of linear constraints, but their model has no way of indicating explicitly that a given region shares a set of line segments (equivalent to linear constraints) with another region. This means that if one updates one of the regions, there will be an inconsistency unless the relationship is somehow stored

explicitly. One can only treat topological relationships indirectly and this makes inconsistencies much more likely.

Another problem is that it is very difficult to find the border of a region from the region representation. Changing the constraints from $x + 2y \leq 5$ to $x + 2y = 5$ is not enough as a conjunction of such constraints is, in general, unsatisfiable. Rather the individual constraints must be converted into line segments, each defined by three constraints (one for the infinite line and two for the end points). The border line is the disjunction of these line segment constraints.

3 Possible Models

In this section three models for storing topological information for moving objects are described and analysed. The first subsection discusses which relationships to store and says something about what needs to be stored for points, lines and regions. The subsequent three subsections describe three different models that might be used to store the shared boundaries given in the first subsection.

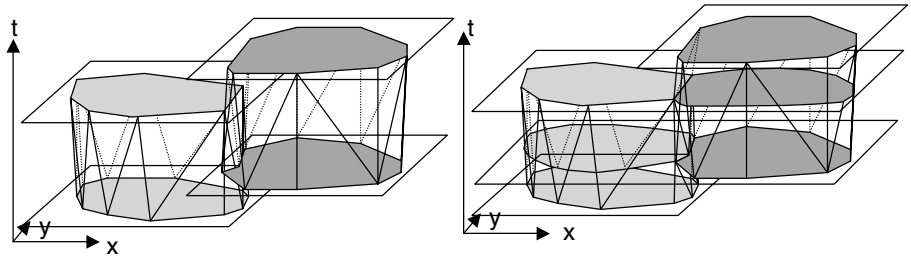
3.1 Which Relationships to Store

The relationships that need to be stored explicitly are those which involve the borders of regions or lines as the borders are infinitely thin and even tiny errors may change the result of the corresponding predicate. The following definitions of the relationships from [4] will be used:

Table 1. Topological Relationships

Operation	Meaning
Disjoint	The two objects do not share either border or interior
Meet	The two objects share borders but not interiors
Overlap	The two objects overlap. This means that the interiors overlap and the borders cross each other
Overlap with disjoint border	The interiors of the two objects oevrlap but the borders do not cross
Cover	One object is inside the other but shares a part of its border
CoveredBy	The reverse of Covers
Inside	One object is entirely inside the other
Contain	The reverse of Inside
Equal	The two objects have equal shapes

For a pair of regions the relationships **meet**, **covers**, **coveredBy** and **equal** involve the border and therefore must be handled explicitly. The other predicates (**overlaps**, **overlaps with disjoint border**, **inside**, **contains** and **disjoint**) deal only with the interiors of the regions and can therefore be computed using the geometry of the objects.



a) Original snapshots of O1 and O2 **b)** Sliced representation of O1 and O2

Fig. 1. Time slices with meeting relationship

For a point and a region, **meet** needs to be stored explicitly while **disjoint** and **inside** can be computed from the geometry. **Meet**[1] can be computed from the geometry if the point moves from outside the region to inside. However, the point may also stay on the border of the region for some time, or it may graze the region. In these last cases the relationship must be stored explicitly or the database might not recognize it.

For a pair of lines both overlap in the end points and interiors must be stored explicitly as in both cases even tiny errors may cause a different answer.

For a point and a line, one must store explicitly any period of time in which the point lies on the line. Crossings can be computed from the geometry.

The only way to identify shared borders reliably is to store the shared part in a shared location like it is done in the Node-Arc-Area representation. For spatiotemporal data, it therefore becomes important to store shared boundaries between spatiotemporal objects.

Line crossings are points and should therefore be stored as a shared moving point. Point crossings may either be a shared moving point (if the point remains on the line or region border over time) or a single spatiotemporal point (with a single time location rather than moving) if the crossing occurs at a particular time instant.

3.2 Time Slices

To maintain the spatial topology in a time slice model, all neighbouring objects must have the same time slices. Inside a time slice the shape of the object is linearly interpolated. Therefore, for each time there is a new snapshot for one of the neighbouring objects, the interpolation of all must change if the borders are to remain equal. Changes in topology should only be allowed between time slices.

The main problem with this approach is that it results in a lot of time slices that are unnecessary for each region but necessary for the whole. One cannot assume that all the regions are updated at the same time (If they were, the time slice approach would work fine). This problem in illustrated in Figure 1.

A partial solution to this problem is as follows: Whenever one object has a snapshot and therefore begins a new time slice, make a new time slice for all neighbouring

[1] A point meets a region when it is on the border of the region.

objects as well, but not objects that are further away. This means that not all objects must have all time slices, but it does mean that each object must have one time slice for each time one of its neighbours is updated as well as for when the object itself is updated. If an object borders four other objects, it will have 5 times as many time slices as if it were isolated.

In the land cover type this problem might be reduced because land cover is usually updated in large areas rather in individual regions. All land cover regions within a given satellite image is probably updated at the same time. Thus this problem only occurs for those land cover regions that lie on the border between different satellite images.

A time slice model in which all objects have the same time slices may use a basic node-arc-area model in each time slice as topology only changes in the instants between time slices.

One problem with a pure time slice approach is handling temporal topological relationships like *enters* and *crosses*. As the time slices of the two objects are probably different, they cannot be used directly as a basis for the topological relationships.

3.3 3D Model

One way of avoiding the problems of time slice models is to use general 3D data types to store moving objects. However, this would require a new model for moving objects as well as a new algorithm for creating moving regions from snapshots.

One drawback of 3D types when compared to a sliced representation is that you lose the direct correspondence between the non-temporal and moving object types. However, one needs only fairly simple operations to extract snapshots from the 3D data types.

Time slices have one other advantage: When you query about a time instant or short time interval, the database only needs to fetch those time slices that are relevant for the query, which may be only a small fraction of the total. For a pure 3D model, on the other hand, the entire geometry of the object needs to be fetched.

3.4 Hybrid Model

This last advantage of time slices may be retained if one defines a hybrid model that looks as follows:

At certain instants in time (typically when there is a snapshot available) the shape of the object is stored. The intervals in between these time instants are time slices in which the shape of the object is inferred. However, rather than using just a linear function to infer the shape, the shape may be represented by any 3D surface that can be represented by a set of 3D triangles and that yields a legal 2D object at all time instants in the time slice. The sliced representation described in [7] would be a special case of this representation.

4 Defining the Hybrid Model

In this section, the hybrid data model is defined in more detail. In Section 4.1, a set of 3D data types is defined. These are then used as building blocks for the hybrid model. The hybrid data types are defined in Section 4.2.

4.1 Building Blocks of the Hybrid Data Types

In this section, data types for 3D points, lines and triangles are defined. Additionally, types for time intervals and temporal line segments are defined.

3D Point: *A 3D point is defined by its coordinates:*

$$3DPoint \equiv \{(x, y, t) | (x \in \Re \wedge y \in \Re \wedge t \in \Re)\}$$

3D Line Segment: Each line segment has two end points:

$$3DLineSegment \equiv \{(s, e) | (s \in 3DPoint \wedge e \in 3DPoint)\}$$

3D Triangle: To make the algorithms for computing intersections easier, surfaces should be piecewise straight. The only way to ensure a piecewise straight surface is to create it as a set of triangles. Therefore, the 3D surface element of this model is a triangle:

$$3DTriangle \equiv \{(p_1, p_2, p_3) | (p_1 \in 3DPoint \wedge p_2 \in 3DPoint \wedge p_3 \in 3DPoint)\}$$

Time Interval: A time interval is a connected set of numbers with a given start and end:

$$Interval \equiv \{(s, e) | (s \in \Re \wedge e \in \Re \wedge s < e)\}$$

Temporal Line Segment: A temporal line segment is a 3D line segment where the end points have ascending times:

$$TempLineSeg \equiv \{s | (s \in 3DLineSegment \wedge s.s.t < s.e.t)\}$$

4.2 Definitions of Spatiotemporal Data Types That Store and Enforce Meeting Relationships

In the hybrid approach outlined in Section 3.4, moving objects are represented by time slices. However, the interpolation between the snapshots is more general than the one given in [7]. For most of the types, all that needs to change is still the definition of the **unit**, or time slice. The overall type can in most cases remain unchanged from the type from [7].

In these definitions, the units are given a semantic meaning of their own. A unit is assumed to be the period between two updates of the object. Thus for every time a moving curve is updated, a new curve unit is added. The same applies to points and regions. Examples of the point and line types are given in Figure 2 and Figure 3. Examples of the face types are given in Figure 4.

4.3 Moving Point

The **moving point** may be modelled as a set of polylines that for each time instant gives only a single point.

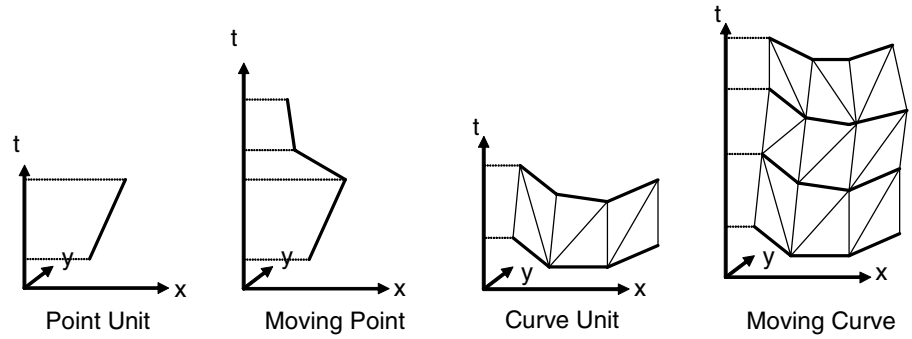


Fig. 2. Moving Point and Curve

The moving point may serve two functions: It may be an *independent* database object in its own right (such as a car, building or marked animal), or it may represent the *meeting point* of several moving curves or regions. These two functions have different storage needs. In this definition, both these types are combined into the single moving point, but it might be argued that they should be different types. They are defined as the same type here to keep the type system small and therefore manageable, and because these two roles are not mutually exclusive.

The following three topological relationships must be dealt with for moving points:

On line: If the point remains on a given moving curve (including the border of a region) over time, this must be stored explicitly as only minor inaccuracies can make the point be outside the line. In most cases the point will be on only one line at a time and for the other case one can conclude that the lines have the same location. The point therefore only needs to store a link to a single line. Since a point is not necessarily on the line during its entire lifetime, this relationship should be stored with the point units rather than in the main point object.

End point of curve: A moving point may serve as a *meeting point* for several moving curves or regions. The most efficient way to store this relationship is to store it in the curves as a curve can have only two end points but a point may be the end point of an arbitrary number of curves. Discovering which curves end in a given point can be done by querying a spatial index with the position of the point. This is guaranteed to return all the curves that end in the point. Any other curves returned can be filtered out by checking their end points.

Meet: Two moving points may have the same position at a particular time instant or in a particular time interval. This relationship may be stored by having the two moving point objects share point units when they are at the same position. Meeting at a time instant can be stored by letting them share a degenerate point unit that is valid only at that time instant.

Temporal meet: Moving points may be connected in time. If for instance one moving point splits into several, there are moving points that meet in time. At the end time of one point it is at the same place as another point object begins.

The **Point Unit** is therefore defined as follows:

$$UPoint \equiv \{(s, c) | s \in TempLineSeg \wedge c \in MCurve\}$$

In this definition, *s* defines the movement of the point and *c* represents the *on line* relationship. An *MCurve* is a moving curve and is defined later.

The **Moving Point** is defined as follows:

$$MPoint \equiv \{(U, M) | U \subset UPoint \wedge M \subset MPoint \wedge \\ \forall(a, b \in U):(a \neq b) \rightarrow \neg tempOverlap(a.s, b.s) \wedge \\ \forall(p \in M):tempMeet(this, p)\}$$

Where *tempMeet(p1, p2)* is true iff the valid time of *p1* meets the valid time of *p2* and the two points are at the same location at that time. *tempOverlap(s1, s2)* is true iff the valid times of *s1* and *s2* overlap.

In this definition, *U* is the set of point units that make up this moving point and *M* is the set of points that *temporally meet* this point.

4.4 Moving Line

The **moving line** type from [9] is defined as a set of moving curves. According to [7], any set of moving line segments makes a valid moving line according to this definition. However, it would be very difficult to deal with topological information with such a construct. It would have no end points, and the border curve of a region is the ideal place to store a pointer to a neighbouring region.

As for a moving point, a moving line may serve two purposes. It may be an *independent* database object in its own right, or it may mark the *border* between two particular regions.

A simple and straightforward definition of a moving curve would be this: A moving curve is a 3D surface consisting of planar facets whose intersection with any plane parallel to the x-y plane would be a valid curve (continuous set of line segments). A moving line could then be defined as a set of moving curves. The moving curve may additionally have to store the following topological information:

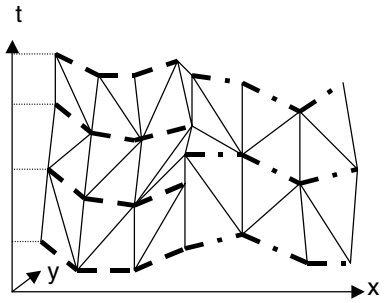


Fig. 3. Moving Line

Bordering regions: The curve may be a part of the border of up to two regions. This is stored in the main curve object. This makes a curve that serves as a *border* represent the border between two particular regions.

End points: Any curve has two end points in space. However, these points only needs to be stored explicitly if they serve as *meeting points* for several curves.

Points on line: If one wants to query which cars are on a particular road, one may want to store which cars are on the road at any given moment. However, this information may take up a lot of storage space and can also be discovered through a spatial search of the points combined with the **on line** relationship for points. It is therefore not necessary to store directly.

Meet: Two *moving line* objects may share *moving curve* objects. If this relationship changes over time, it is handled the same way as two regions that stop bordering one another.

Temporal meet: If two area objects that used to border each other no longer do, then the border curve should split into two new curves, one for each area object. These new curve objects should store the fact that they are continuations of an old curve.

The **curve unit** can be defined as follows:

$$UCurve \equiv \{(vt, T) | vt \in Interval \wedge T \subset 3DTriangle \wedge \forall (t \in vt):(AtInstant(t, T) \in Curve)\}$$

The *AtInstant* function creates the intersection between a flat plane at a given time and a given set of 3D objects. The *Curve* type represents a non-temporal curve.

The **moving curve** is a set of curve units:

$$\begin{aligned} &MCurve \equiv \{(C, e1, e2, f1, f2, TM) | C \subset UCurve \wedge \\ &e1 \in MPoint \wedge e2 \in MPoint \wedge \\ &f1 \in MFace \wedge f2 \in MFace \wedge TM \subset MCurve \wedge \\ &\forall (a \in C) \forall (b \in C):(Overlap(a.i, b.i) \rightarrow (a = b)) \wedge \\ &endPoint(e1) \wedge endPoint(e2) \wedge \\ &borderFace(f1) \wedge borderFace(f2) \wedge \\ &\neg Overlap(f1, f2) \wedge \\ &\forall (mc \in TM):TempMeet(this, mc)\} \end{aligned}$$

In this definition, *endPoint*(*p*) indicates that the point is an end point of this curve, and *borderFace*(*f*) indicates that the face is bordered by the curve. The *MFace* type represents a moving face and is defined later.

The **moving line** is a set of moving curves. It does not require that those curves have the same time slices. The reason for this is given in the next section. In the example in Figure 3 the different dash patterns indicate different moving curves.

$$MLine \equiv \{C | C \subset MCurve\}$$

4.5 Moving Region

The **moving region** type from [9] is defined as a set of non-overlapping faces. A face is a connected area object that may have any number of holes.

The main topological relationship between faces that should be handled explicitly is **bordering**, that is, which regions border this region. This relationship can be deduced from the *bordering regions* relationship for moving lines.

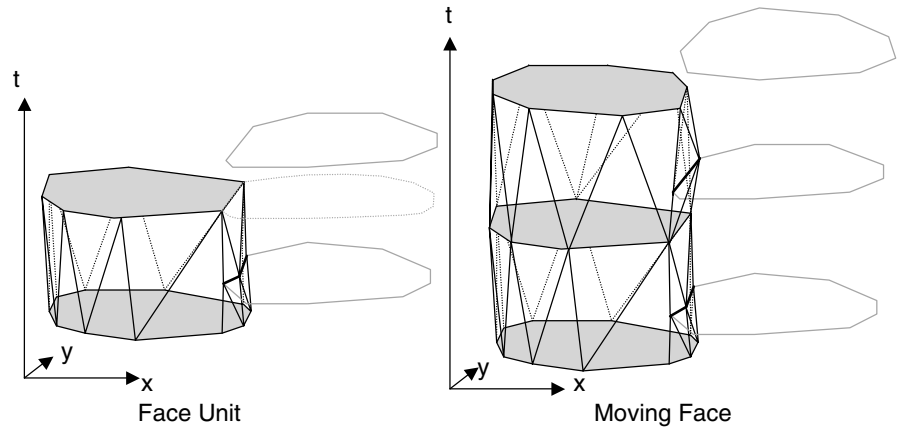


Fig. 4. Moving Face

A **moving cycle** is a set of moving curves that for any time instant in the time period that the cycle object is valid forms a ring. In the models from [9], a moving cycle is considered to consist of only one curve. However, when one wants to store topology it is better to think of a cycle consisting of a set of curves. Each curve in the set represents the boundary between two particular regions. The moving cycle is therefore defined based on the *moving line* type.

$$MCycle \equiv \{(l, vt) | l \in MLine \wedge vt \in Interval \wedge \forall (t \in vt): AtInstant(t, l) \in Cycle\}$$

The curve units in each moving curve represent the boundary between two particular region units. Since the regions bordering a particular cycle may be updated at different times and therefore have snapshots at different times, the curve units in each curve in an line must be allowed to have different time slices.

A moving face consists of one moving cycle representing the outer boundary of the face and N moving cycles defining the holes. Discrete changes in the moving face are assumed to happen in the instants between time slices. All the cycles in a given face unit should be valid in the time that the face unit is valid because changes in topology such as the appearance of new holes should only occur between time slices. The **face unit** is therefore defined as follows:

$$UFace \equiv \{(oc, HC, vt) | oc \in MCycle \wedge HC \subset MCycle \wedge vt \in Interval \wedge \forall (hc \in HC): (Inside(hc, oc) \wedge (hc.\mathrm{vt} \supseteq vt)) \wedge (oc.\mathrm{vt} \supseteq vt)\}$$

In this definition, *Inside*(*a* ,*b*) is true iff *a* is inside *b*.

Notice that in this definition the face units may be different from the units of the moving curves. This is because a curve has a new unit whenever one of the faces that it borders is updated. Thus if curve *a* borders faces *b* and *c*, *a* has a number of units equal to the total number of units of *b* and *c*.

A **moving face** is defined as a non-overlapping set of face units:

$$MFace \equiv \{UF | UF \subset UFace \wedge \\ \forall(a \in UF)\forall(b \in UF):(Overlap(a.\text{vt}, b.\text{vt}) \rightarrow (a = b))\}$$

A **moving region** is a set of moving faces. These are not required to have the same time slices. If a region consists of several faces one is not guaranteed that snapshots of all the faces from the same time exist.

$$MRegion \equiv \{F | F \subset MFace \wedge \\ \forall(a, b \in F):(a \neq b \rightarrow Disjoint(a, b))\}$$

5 Constructing the Hybrid Model

This section presents a method for constructing the hybrid model of moving regions from a series of snapshots of the individual regions. This method assumes that the regions are updated periodically but not necessarily simultaneously. The topological relationships between the regions are stored in an adjacency graph. All topological relationships should be maintained unless this graph is explicitly changed.

Modelling a partition and modelling a network are two sides of the same coin as the borders between the regions in a partition forms a network. A model for one also works for the other. This also applies to partial partitions (ones that cover only a part of the space of interest).

5.1 Constructing a Moving Partition

When creating the component objects of a moving partition, it is easier to interpolate the border curves than the regions themselves. One approach based on regions would be to interpolate each region separately and then ensure that their borders meet. Writing a procedure that could do this and ensure consistency in places where more than two regions meet would be quite complex. Therefore, the algorithm presented here for creating a moving partition is based on interpolating the border curves rather than the regions. The algorithm is also based on an adjacency graph[2] supplied by the cartographer. This adjacency graph must be explicitly updated when the topology changes.

The algorithm assumes that there is a pre-existing partition that one wants to update. When creating a partition for the first time, one runs a similar algorithm for each cycle in the adjacency graph of the initial partition as well as for each edge that does not belong to a cycle.

When modifying the regions so that they fit together in the partition, the system always stores the original versions as well as the modified ones. The original versions are used for interpolations whenever a new version of a region is inserted. This is dome to ensure that the interpolated versions of the regions stay as close to the original as possible, especially if one region is updated several times while another is not.

[2] An undirected graph with one node for each face and an edge between each pair of faces that border each other.

Algorithm. *UpdatePartitionInterpolation*(*nf*, *FS*, *ag*)
Input: A new face snapshot *nf*, the set of faces in the partition *FS*, and an adjacency graph for the faces *ag*.
Output: The face set with a new snapshot added
Method:
Let *of* be the previous snapshot of *nf*
Let *fn* be the node in *ag* that represents *of*
Let *mf* be a copy of *nf*
For each cycle in *ag* that contains *fn* **do**
Compute a meeting point for all the faces in the cycle using the original rather than the modified snapshots. This is done by adding a buffer of the same size for all the faces (if there is a gap) or subtracting an area of the same size for all the faces (if they all overlap).
End for
For each edge in *ag* that ends in *fn* **do**
Construct a meeting line between the two faces by adding a buffer to the other face and removing overlap until they meet. For each side where there is a cycle ensure that the lines end in the meeting points.
Update *mf* by replacing the original line with the new meeting line (See Section 5.2)
End for
Interpolate the lines of the face *mf* from their version in *of*
Add the interpolation and *mf* to *FS*
Add *nf* to *FS* as the original face (for use in later runs of this algorithm)
return *FS*
End *UpdatePartitionInterpolation*

The adjacency graph is supplied by the cartographer who knows which regions are supposed to border each other. Updates to the topology is reflected in updates of the adjacency graph. The following updates are possible:

- Removing an edge
  - At the edge of the partition: This region no longer borders that region. The interpolation system should ensure that from the point in time in which the edge was removed there is a small gap between these two regions. (Regions in the adjacency graph should not overlap. A region may be added to the graph as an isolated node to indicate that it should not overlap any of the other regions).
  - In the middle of the partition: These two regions no longer border each other. Reduce the meeting line to a point at the time in which the edge was removed. This point may then expand into a line between a new pair of regions. At the instant when the line is removed neither it nor any newly inserted lines exist thus forming a possibly temporary cycle.
- Removing a node
  - As nodes correspond to faces, this means that a face no longer exists. Insert a single point representing the face as a new version and interpolate neighbouring faces to this point. This point is the new meeting point of the eventual new cycle created by removing the node. All the edges from that node are also removed.

Adding an edge or node is just like removing one except that the process is reversed in time.

Whenever the adjacency graph is updated, all affected faces must have a new time slice, where the state at the beginning of this new time slice is the state according to *UpdatePartitionInterpolation* after the change in the graph.

5.2 Constructing a Moving Network

A network may consist either of nodes (points) with curves between them (like graphs except that the shape of the curves may be important) or routes and intersections. The difference between these two is that the routes-and-intersections model can be equivalent to a non-planar graph as routes may cross each other without intersecting (one road goes in a tunnel below another with no means of switching from one to the other).

If the entire network is updated at the same time instants, representing a changing network is trivial. A network unit is simply a collection of moving curves which end in the same moving points (nodes). Changes in topology (removal of edges or nodes, merging of nodes and edges) happen only between the time slices (at the end of one and beginning of the next) in this model.

When the curves are updated individually, they are interpolated as normal. Additionally, a new version of the end points are stored. The end points are interpolated in time between all the end points of all the curves that meet there. Thus the meeting point of four curves would be updated whenever any of the four curves is updated. This means that the final line segment of each curve changes more often than the other lines, and the interpolation of these line segments cannot use the normal algorithm from [12]. One alternative method would be this:

- Create the projection of the final line segments and the changing end point on a plane that is parallell to the time axis and has an angle to the x-y axis that is the average of the angle of the final line in the two snapshots.
- Create the Delaunay triangulation of the projected lines.
- Use this as the triangulation of the final line segments. Going back to a full 3D representation is easy since the triangulation does not insert any new points and the 3D coordinates of the points used are known.

One example of such a curve and the proposed interpolation algorithm is shown in Figure 5.

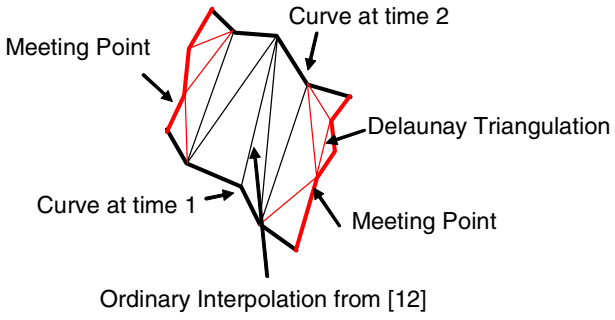


Fig. 5. Interpolating a line in a network

For networks consisting of routes and intersections, a modified version of this method can be used. Each route is equivalent to a connected set of edges and each intersection is equivalent to a meeting point. Places where routes cross but there is no intersection are not represented as points. This is a simple way of distinguishing such crossings from regular intersections.

6 Handling Current Time

The database described so far handles historical spatiotemporal data quite well. However, when asked about the current state, it can only return the last state of the various polygons, and this state can be inconsistent for those polygons which have not been updated for some time. The most straightforward method is to assume that the objects are static after the last update. This method works well if the objects do not move too much. The method basically goes as follows:

- Take the most recent snapshot in the area of interest.
- Insert a new time slice of each object that extends from the last snapshot of the object to the current time. The object is considered to be static in this time slice.
- In this new time slice, all the moving points and curves are static and do not change from their most recent state. This includes the curves that form the boundaries of moving regions.

This is a fairly simple method that works well in many cases. However, it does not take the movement of the objects into account and may therefore produce highly inaccurate results in cases in which the objects move fast.

One alternative is to try to extrapolate them based on past movement. To do this for a general line or region is quite complex. A naïve approach would be to extrapolate based on the movement of the individual points that make up the line or region border. This produces artifacts like shown in Figure 6.

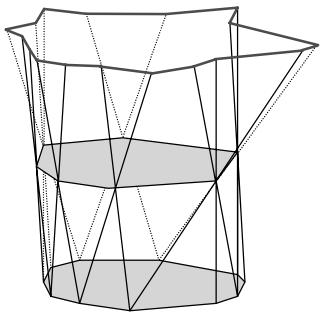


Fig. 6. Extrapolating using a Triangle Representation

7 Implementing the Model

In this model many objects should have references to each other. For instance, the region object stores its boundary as a set of curve objects. This ensures consistency,

but it also creates a problem. To fetch the geometry of a region, one must access all the curve objects that make up its boundary, and all the point objects that make up the meeting points for those curves. Unless these are stored together, this may slow down the system. Border curves cannot be stored together with both of the regions that they border as these regions may be stored in entirely different places on the disk.

An alternative is to store several copies of these objects. For the curve, one would store one copy for each of the two regions that have it as part of their boundaries. This would make retrieval more efficient, but would introduce the possibility of inconsistencies. To solve this one would need a database system that could handle integrity rules of the type "Objects A and B should always be equal". Such a system could then enforce this by always updating both curves when one is updated. This would increase the cost of updating as well as storage cost, but would reduce query costs.

For many of the relationships, such as a point serving as an end point, the relationship is stored in only one object, typically the object for which the relationship has the lowest cardinality. This is common design practise in relational databases and helps to ensure consistency. However, one may choose to store the relationships both ways instead. This will increase storage cost as well as update cost to avoid inconsistencies, but may reduce query cost.

[6] defines two relationships as examples of temporal topological relationships:

- **Enters:** The object starts outside the region and moves inside it.
- **Crosses:** The object enters the region and later leaves it.

The time slices in the hybrid model may be used to reduce the amount of data that needs to be fetched to do these computations. If each time slice has its own spatio-temporal bounding box, an index may be used to find those time slices of each object in which they may overlap. Then only those time slices need to be used to compute the temporal topology as for all other time slices the objects are known to be **disjoint**.

8 Summary

This paper has presented an extension to the sliced representation from [7] that is capable of representing explicit topology. This is necessary for several important operations, including checking whether two regions border each other. The new representation is slightly more complex than the original representation, but many of the same algorithms are applicable to both.

References

1. J. Chomicki and P. Z. Revesz: Constraint-Based Interoperability of Spatiotemporal Databases. In *Proc. 5th Int. Symp. on Large Spatial Databases*, pp. 142-161, 1997.
2. J. Chomicki and P. Z. Revesz: A Geometric Framework for Specifying Spatiotemporal Objects. In *Proc. 6th Int. Workshop on Temporal Representation and Reasoning (TIME'99),* pp. 41-46, 1999\

3. M. J. Egenhofer and K. K. Al-Taha: Reasoning about Gradual Changes of Topological Relationships. In A. Frank, I. Campari, and U. Formentini (eds.): *Theory and Methods of Spatio-Temporal Reasoning in Geographic Space*, vol. 639 LNCS, Springer-Verlag, pp. 196-219. 1992
4. M. J. Egenhofer and R. D. Franzosa: Point-Set Topological Spatial Relations. In *Int. Journal of Geographical Information Systems*, 5(2), pp. 161-174, 1991
5. M. Erwig and M. Schneider: The Honeycomb Model for Spatio-Temporal Partitions. In *Proc. Int. Workshop on Spatio-Temporal Database Management*, pp. 39-59, 1999
6. M. Erwig and M. Schneider: Spatio-Temporal Predicates. In *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 14(4), pp. 881-901, 2002
7. L. Forlizzi, R. H. Güting, E. Nardelli, M. Schneider: A Data Model and Data Structures for Moving Objects Databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data* (Dallas, Texas),pp. 319-330, 2000
8. S. Grumbach, M. Koubarakis, P. Rigaux, M. Scholl, S. Skiadopoulos: Spatio-Temporal Models and Languages: An Approach Based on Constraints. In *Spatio-Temporal Databases - the CHOROCHRONOS Approach*, LNCS 2520, Springer-Verlag, pp. 177-201, 2003
9. R. H. Güting, M. F. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, M. Vazirgiannis: A Foundation for Representing and Querying Moving Objects. In ACM Transactions on Database Systems 25(1), 2000.
10. A. Renolen: Concepts and Methods for Modelling Temporal and Spatiotemporal Information. *Dr. Ing Thesis,* Norwegian University of Science and Technology (NTNU), 1999
11. M. Schneider: Implementing Topological Predicates for Complex Regions. In *Proc. 10th Int. Symp. on Spatial Data Handling (SDH),* pp. 313-328, 2002
12. E. Tøssebro and R. H. Güting: Creating Representations for Continuously Moving Regions from Observations. In *Proc. 7th Int. Symp. on Spatial and Temporal Databases*, pages 321-344, 2001
13. M. Worboys: *GIS: A Computing Perspective.* Taylor & Francis, 1995