Creating Representations for Continuously Moving Regions from Observations

Erlend Tøssebro[1] and Ralf Hartmut Güting[2]

Abstract

Recently there is much interest in moving objects databases, and data models and query languages have been proposed offering data types such as *moving point* and *moving region* together with suitable operations. In contrast to most earlier work on spatio-temporal databases, a moving region can change its shape and extent not only in discrete steps, but continuously. Examples of such moving regions are oil spills, forest fires, hurricanes, schools of fish, spreads of diseases, or armies, to name but a few.

Whereas the database will contain a "temporally complete" representation of a moving region in the sense that for any instant of time the current extent and shape can be retrieved, the original information about the object moving around in the real world will most likely be a series of observations ("snapshots"). We consider the problem of constructing the complete moving region representation from a series of snapshots. We assume a model where a region is represented as a set of polygons with polygonal holes. A moving region is represented as a set of *slices* with disjoint time intervals, such that within each slice it is a region whose vertices move linearly with time. Snapshots are also given as sets of polygons with polygonal holes. We develop algorithms to interpolate between two snapshots, going from simple convex polygons to arbitrary polygons. The implementation is available on the Web.

1 Introduction

Databases have for some time been used to store information on objects which have positions or extents in space. There are also many applications of databases which store information about how such objects change over time. Spatial objects that move or change their shape over time are often referred to as moving objects. In [BGE+00] an abstract model for representing moving objects in databases is described. In an abstract model, geometric objects are modeled as point sets. For continuous objects like lines or regions, these point sets are infinite. This means that these models are conceptually simple, but cannot be directly implemented. A discrete model, on the other hand, can be implemented but is somewhat more complex. A discrete model for spatio-temporal objects, which builds on the abstract model in [BGE+00], is described in [FGNS00].

Early research on spatio-temporal databases concentrated on modeling discrete changes to the database. Examples of such models can be found in [W94], [CG94], and [PD95]. More recent research also addresses the dynamic aspect, that is, that objects may change continuously without explicit updates. One example of such a model is presented in [SWCD97]. However, this model covers only the current and expected near future of the objects, and not the histories of the objects, and it also does not deal with moving regions. Constraint databases can also be used to describe such dynamic spatio-temporal databases. One study of constraint databases which explicitly addresses spatio-temporal issues is [CR97]. [CR99] contains a framework in which all spatio-temporal objects are described as collections of *atomic geometric objects*. Each of these objects is given as a spatial object and a function describing the development of this object over time. For the continuous functions, affine mappings (allowing translation, rotation and scaling) and subclasses of these are considered. However, to the authors'

1. Department of Computer Science, Norwegian University of Science and Technology, N-7491 Trondheim, Norway, tossebro@idi.ntnu.no
2. Praktische Informatik IV, Fernuniversität Hagen, D-58084 Hagen, Germany, gueting@fernuni-hagen.de

knowledge, [BGE+00] and [FGNS00] describe the only comprehensive model describing spatio-temporal data types and operations.

The model in [FGNS00] describes a way to represent continuously moving, amorphous objects in a database in such a manner that it is possible to produce a "snapshot" of the object at any time within the time interval in which it exists. However, most data about moving objects will come in the form of snapshots taken at specific times. This paper addresses the problem of creating this type of representation from a series of snapshots of a moving amorphous region. Important types of such regions in the real world would be oil spills, forest fires, fish schools, and forests. (Forests change continuously because of deforestation, climatic changes, etc.).

This problem is similar to the problem of interpolating or blending shapes, which has been studied in the computer graphics community, because both problems involve creating plausible in-between shapes at any time between the two states given. One example of such a shape interpolation algorithm is given in [SG92]. This algorithm was created to solve the problem of creating a smooth blending between two figures in an animated movie. A comparison between the algorithm given in [SG92] and our algorithm is given in Section 8.

A problem which occurs when the moving region consists of several disjoint parts is to discover which part in the first snapshot corresponds to which part in the second snapshot. Because the region parts may have changed both their positions and shape, it may not be obvious to a computer which of them to match. One region part may also have split into two between the two snapshots.

In Section 2 the representation of regions and moving regions from [FGNS00] is described. Section 3 then introduces the basic algorithm for building this representation for convex regions. In Section 4, a way of representing a non-convex area as a tree of convex areas with convex concavities is described. This structure is later used to apply the technique described in Section 3 for non-convex regions. Section 5 describes strategies for discovering which regions, or components of regions, in one snapshot correspond to which regions in the other snapshot. This is important both for creating representations for multi-component regions and for matching parts of the tree representation of Section 4 correctly. Section 6 describes the algorithm for interpolating between arbitrary polygons; an important subproblem is the matching of concavities between snapshots. In Section 7, the quality of the results for different types of regions is discussed. Section 8 is a comparison between our work and [SG92], and Section 9 contains the conclusions to this paper.

2 Representing Regions and Moving Regions

In this section we review the structure and representation of static and moving regions defined in [FGNS00], since this representation needs to be created by our algorithms. We start by considering a (static) region, as a moving region needs to be consistent with it. Indeed, a moving region, evaluated at any instant of time, yields a region.

A *region* may consist of several disjoint parts called *faces*, each of which may have 0 or more holes. At the discrete level, the boundaries of faces as well as holes are described by polygons. Hence a region looks as shown in Figure 1.

This structure is defined in terms of *segments*, *cycles*, and *faces*. We sketch the structure of the formal definitions in [FGNS00]; more details can be found there.

$$Seg = \{(u, v) \mid u, v \in Point, u < v\}$$

A *segment* is just a line segment connecting two points which need to be distinct.

$$Cycle = \{ S \subset Seg \mid ...\}$$

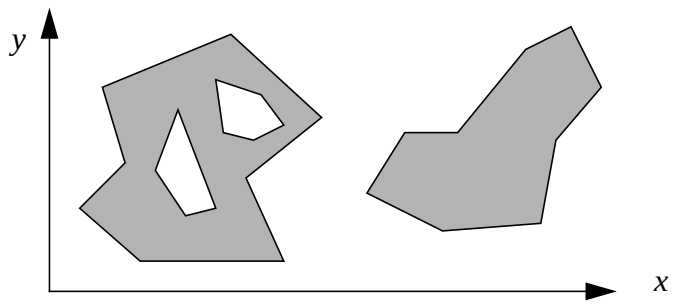


Figure 1: A region

A *cycle* is a set of line segments forming a closed loop which does not intersect itself, hence it corresponds to a simple polygon.

$Face = \{(c, H) \mid c \in Cycle\ ,\ H \subset Cycle\ ,$ such that ...$\}$

A *face* consists of a cycle *c* defining its outer boundary, and a set of cycles *H* defining holes. These holes must be inside the outer cycle, and must be pairwise disjoint. *H* may be empty.

$$Region = \{ F \subset Face \mid f_1, f_2 \in F \wedge (f_1 \neq f_2) \Rightarrow edge\text{-}disjoint\ (f_1, f_2) \}$$

A *region* is a set of disjoint[1] faces.

A *moving region* is described - like the other “moving” data types in [FGNS00] - in the so-called *sliced representation*. The basic idea is to decompose the temporal development of a value into fragments called *slices* such that within a slice this development can be described by some kind of “simple” function. This is illustrated in Figure 2.

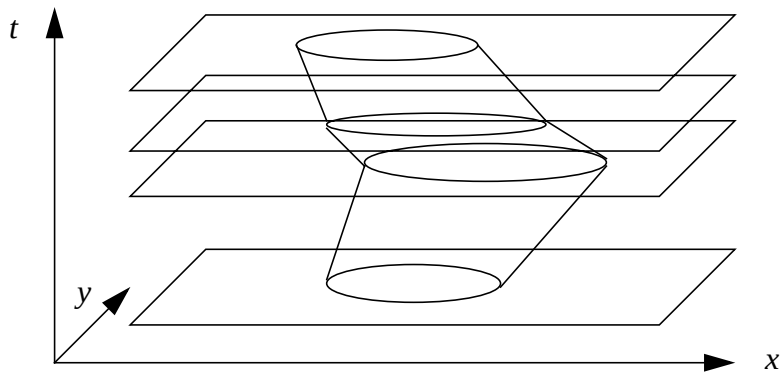


Figure 2: Sliced representation

Hence each slice corresponds to a time interval; the time intervals of distinct slices are disjoint. For a moving region, the “simple function” within a single slice is basically a region (as defined above) whose vertices move linearly in such a way that at any instant of time within the slice a correct region is formed. Such a slice is shown in Figure 3.

1. Edge-disjoint means that two faces may have common vertices, but must otherwise be disjoint (i.e., they may not share edges).

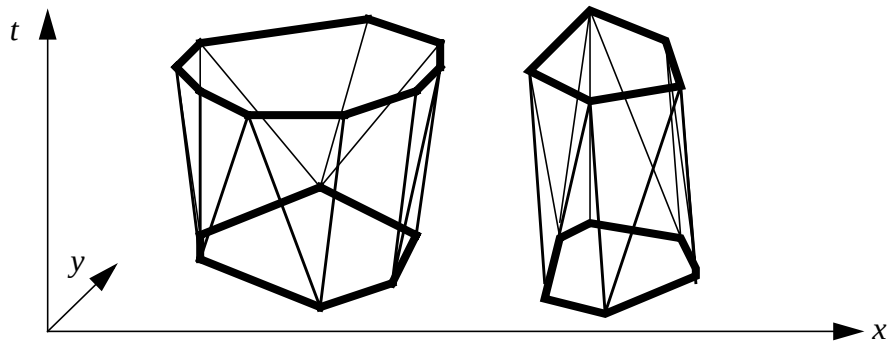


Figure 3: A slice of a moving region representation

The structure represented within a single slice of a moving region is called a *region unit*. This structure is defined bottom-up in terms of *moving points*, *moving segments*, *moving cycles*, and *moving faces* analogously to the definition of a region. Again we sketch the formal definitions from [FGNS00].

$$MPoint = \{(x_1, x_1, y_0, y_1) \mid x_0, x_1, y_0, y_1 \in real\}.$$

A moving point is given by four real coordinates. The semantics of this four-tuple, that is, the function for retrieving the position of the moving point at any point in time is

$$p(t) = (x_0 + x_1 \cdot t, y_0 + y_1 \cdot t)$$

In the three-dimensional (*x, y, t*)-space, a moving point forms a straight line.

A moving segment is defined by:

$$MSeg = \{(s, e) \mid s, e \in MPoint, s \neq e, coplanar(s, e)\}$$

A moving segment consists of two moving points which are coplanar, i.e., lie in the same plane in the (*x, y, t*)-space. Hence in 3D a moving segment is a trapezium (Figure 4a). The segment may degenerate at one end of the

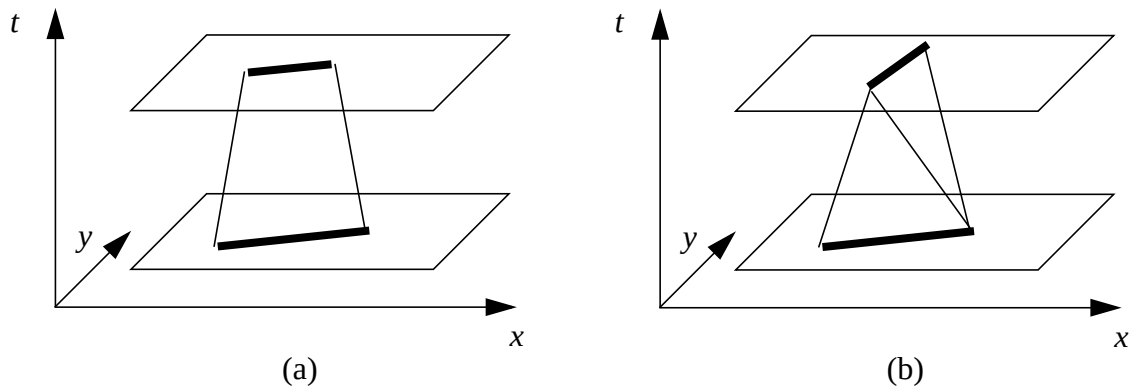


Figure 4: (a) A moving segment. (b) Two moving segments representing a rotating line segment.

time interval into a point, hence we may have a triangle in the 3D space. This means that a moving segment cannot rotate as time passes. One can create a (rough) representation for a line segment which rotates by creating two moving segments, each of which is the line segment in one snapshot and becomes a point in the other (Figure 4b).

$$MCycle = \{(s_0, ..., s_{n-1}) \mid n \geq 3, s_i \in MSeg\}$$

An *MCycle* is the moving version of the *Cycle*. It contains a set of moving line segments. None of these may intersect in the interior of the time interval in which the *MCycle* is valid. The *MCycle* must yield a valid *Cycle* in all instants in the interior of the time interval.

$$MFace = \{(c, H) \mid c \in MCycle, H \subset MCycle\}$$

This is a moving version of the *Face*. The *MFace* must yield a valid *Face* in all time instants in the interior of the time interval.

$$URegion = \{(i, F) \mid i \in Interval, F \subset MFace \text{ such that } ...\}$$

A region unit consists of a time interval and a set of moving faces such that evaluation at any instant of time in the interior of the time interval yields a valid region value.

3 The Easy Case: Interpolating Between Two Convex Polygons

The problem is now to compute from a list of region snapshots a moving region representation. This reduces to the problem of computing a region unit from two successive snapshots.

In this section we first consider the most simple case of the problem which occurs if each of the two snapshots is a single convex polygon without holes. In this case one can apply an algorithm that we call the “rotating plane” algorithm. It can be described as follows. Input are two convex cycles at different instants of time.

To create one moving segment, start with a segment *s* in one of the polygons and create a plane perpendicular to the time axis through it. Then rotate that plane around segment *s* until it hits a segment or a point from the other polygon[1]. If in the other polygon there exists a segment *s*’ which is parallel to *s*, then the plane will hit this segment, and the algorithm will create a proper trapezium-shaped moving segment between *s* and *s*’. If there is no parallel segment, then the plane will hit a point *p*. Then a degenerate moving segment will be created which starts out as the original segment *s* and ends as point *p*, thus forming a triangle in space-time.

This algorithm can be implemented in a computer in the following fashion: Take the segments in both polygons and sort them according to their angle with respect to the *x*-axis (for instance). Then go through the two lists in parallel, starting with the segment with the smallest angle in either list. For a given segment check the next segment in the other list. If the angle of this segment is equal to the angle of the chosen segment, create a proper moving segment connecting the two and mark both segments as done. If the angle is different, take the first point in the other segment, use it as the second “segment”, and mark only the chosen segment as done. After the moving segment is formed, take the unmarked segment from either list with lowest angle as the next segment.

An example of the matchings generated by this algorithm is given in Figure 5. Because the angle of segment *c* is greater than the angle of segment *a*, and less than the angle of *b*, the segment *c* is matched to the point between segments *a* and *b.*

We now give a more formal description of this algorithm (Figure 6). The representation of a line segment (*Seg*) is extended to contain an angle as well as the two end points. Also a function *make_moving_point* (Figure 7) is used to create a moving point from two static points.

Computing the angles between all segments and the *x*-axis takes O(*n*) time, where *n* is the total number of segments. Finding the segments with the lowest angle can also be done in O(*n*) time. Assuming the segments in the two snapshots are already ordered so that adjacent segments are also neighbours in the list, finding the next seg-

1. It should be rotated in such a direction that the part which moves towards the other object hits the other object on the same side as the segment is on the first object.

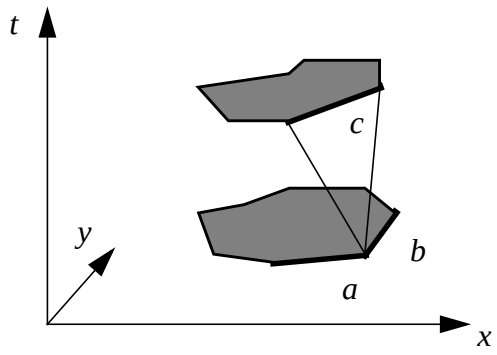


Figure 5: Example of matching created by the rotating plane algorithm

---

algorithm *rotating_plane*(s_1, s_2, t_1, t_2)
input: Two convex *cycles*, s_1 and s_2, which represent snapshots of the moving cycle at the distinct times t_1 and t_2, respectively.
output: An *mcycle* which yields the two *cycles* at the given times.
method:
let $s_1 = \{s_{1,1}, ..., s_{1,n}\}$; let $s_2 = \{s_{2,1}, ..., s_{2,m}\}$;
let *um* be a list of *Segs*; *um* := $\emptyset$;
for each $s_{i,j}$ **do**
compute the angle between $s_{i,j}$ and the *x*-axis, and store it in $s_{i,j}$.*angle*;
um := $um \cup \{s_{i,j}\}$
end for;
MCycle r := $\emptyset$;
while ($um \neq \emptyset$) **do**
l_1 := the $s_{i,j}$ with the lowest angle, $s_{i,j} \in um$;
l_2 := the $s_{k,l}$, $k \neq i$, with the lowest angle, $s_{k,l} \in um$;
if no such l_2 exists **then**
l_2 := the $s_{k,l}$, $k \neq i$, with the lowest angle
end if;
if ($l_1 \in s_1$) **then**
let $l_1 = (a, b)$; let $l_2 = (c, d)$
else let $l_1 = (c, d)$; let $l_2 = (a, b)$
end if;
let mp_1 and mp_2 be *MPoints*;
if (angle of l_2) = (angle of l_1) **then**
mp_1 := *make_moving_point*(a, c, t_1, t_2);
mp_2 := *make_moving_point*(b, d, t_1, t_2);
um := $um \setminus \{l_1, l_2\}$
else
mp_1 := *make_moving_point*(a, c, t_1, t_2);
mp_2 := *make_moving_point*(b, c, t_1, t_2);
um := $um \setminus \{l_1\}$
end if;
MSeg ms := (mp_1, mp_2);
r := $r \cup \{ms\}$
end while;
return *r*
end *rotating_plane*

Figure 6: Algorithm *rotating_plane*

function *make_moving_point*(*a*, *b*, t_0, t_1)
input: Two points, *a* and *b*, and two distinct times t_0 and t_1.
output: A moving point which is at *a* at time t_0 and at *b* at time t_1.
method:
    dx := (*b.x* - *a.x*) / (t_1 - t_0);
    dy := (*b.y* - *a.y*) / (t_1 - t_0);
    mp := (*a.x* - *dx* · t_0 , *dx*, *a.y* - *dy* · t_0 , *dy*);
    return(*mp*)
end *make_moving_point*

Figure 7: Function *make_moving_point*

ment with the lowest angle can be done in constant time (test the next segments in both snapshots and use the smaller one). Adding a new moving segment to the result *r* can also be done in constant time. Because both of the last two operations must be performed once for every segment, the total time for them is O(*n*). Therefore, this algorithm takes O(*n*) time. Note that in the implementation the removal from *um* and checking for membership in *um* is done by modifying or checking a variable associated with each line segment rather than by physically removing or checking in a set. This also applies to the other algorithms below which use a set of unmarked objects.

If the segments are unsorted or sorted by a different criterion than ordering along the border of the cycle, sorting them by angle takes $O(n \cdot \log(n))$ time, and hence the running time of the algorithm will grow to $O(n \cdot \log(n))$.

Theorem 1: Given two convex cycles c_1 and c_2 at times t_1 and t_2, algorithm *rotating_plane* computes a region unit connecting these two cycles. If the two cycles consist of a total of *n* segments and the cycles are represented in (e.g. clockwise) order, then the algorithm requires O(*n*) time. If the two argument cycles are not given in order, then $O(n \cdot \log(n))$ time is required.

A problem with this interpolation method is that it is poor in handling rotation. If a long, thin object rotates 90 degrees between snapshots, the interpolation in the middle between them will be more or less quadratic, and will probably have a much larger area than the object has in either snapshot. For this reason, one must ensure that the snapshots are so close to each other in time that only a small amount of rotation has happened between them.

So far we can handle a single convex polygon in both snapshots, the most simple case. Two major problems remain:

1. Treating concavities.
2. Treating regions with more than one face. Here the problem is to match faces from the first snapshot correctly with faces from the second snapshot. Another version of this problem is one face with several holes. One face with one hole can be treated by interpolating separately between the outer cycles from the two snapshots and the two hole cycles and then subtracting the “moving hole” from the “moving outer cycle”. But if there are several holes, the algorithm must discover which holes correspond.

These problems are addressed next.

4 Representing Non-Convex Polygons by Nested Convex Polygons

We now focus on treating a region which still consists of a single face without holes, i.e., a single cycle, but which needs not be convex any more. The basic idea is to reduce this problem to the previous one by viewing a non-convex polygon as being composed recursively from convex components.

This section first describes a representation in which a general cycle is stored as nested convex polygons. The second subsection describes an algorithm for generating this representation from a *Cycle*.

4.1 The Convex Hull Tree

This is a way to store arbitrarily shaped regions by storing convex regions with convex holes. These convex regions and convex holes may then be treated independently by the rotating plane algorithm, allowing it to work for objects with concavities as well.

In an abstract view of the convex hull tree, each node *p* represents a convex cycle *c* without holes. Each descendant *d* of *p* represents a hole to be cut out from *c* to form the cycle represented by the subtree rooted in *d*. This general method may be used both for storing real holes and for storing concavities in the object. A concavity can simply be represented by a hole which includes a part of the boundary of the cycle. See Figure 8.

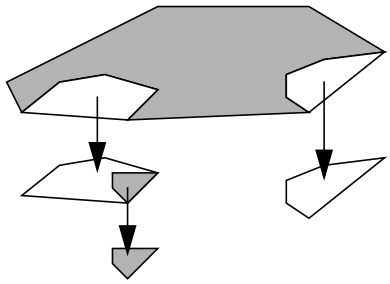


Figure 8: A convex hull tree

In the implementation of the convex hull tree, a cycle is stored in the following manner: Each node contains a list of line segments representing the convex hull of the cycle. For each of the segments in this representation which were added to make the cycle convex, a link to a child node is stored. This child contains the convex hull of the area which should be extracted to get the real cycle. If the extracted area contains concavities itself, then the child will have children of it’s own with extracted areas.

An example of a cycle with several concavities and a convex hull tree representation of this cycle is shown in Figure 9. In this figure, the cycle itself is represented by the thick lines. The segments of medium thickness were added to make it convex. The other segments were added to make nodes further down in the tree represent convex areas. The top node of the tree representation to the right in the figure contains the segments of the convex hull. The line style is the same in the nodes as in the drawing of the region.

This structure as it is described here cannot store holes, because a hole is not connected to a segment in the parent node. However, one could permit the root[1] node to have links to subnodes which are not connected to any particular segment. These would then represent holes.

4.2 Computing a Convex Hull Tree from a Polygon

To build a convex hull tree for an arbitrarily shaped polygon, use the following steps:

1. Start at the root node and the entire polygon.

1. This should not be permitted for nodes other than the root. If the hole is in the object itself, it should be linked to the root. If the hole is in a concavity, then the object is no longer a single region, but several disjoint regions.

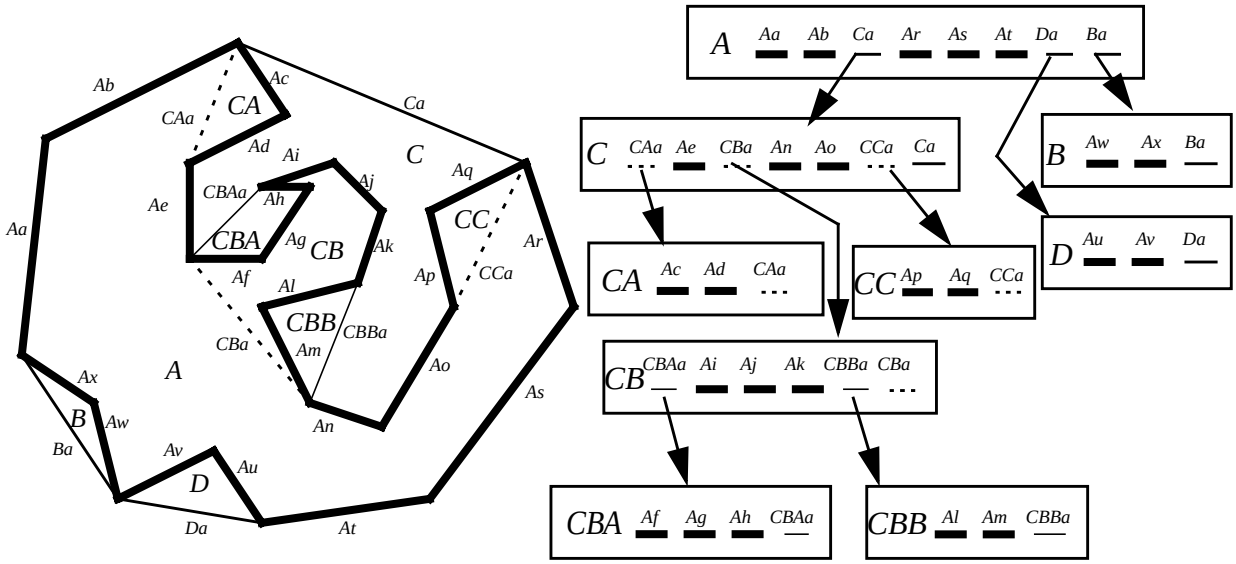


Figure 9: A region with concavities and its convex hull tree representation.

2. Create the convex hull of the polygon.
3. Store a segment list representation of the convex hull into the node.
4. For each of the segments which were added to make the polygon convex, create a new node.
5. For each of these holes with new nodes, go to step 2.

The algorithm for building a convex hull tree (Figure 10) uses two new types, *CHTNode* and *CHTLineSeg*. *CHTLineSeg* is a line segment (*Seg*) which in addition to the two end points may store a link to a child *CHTNode*. The *CHTNode* type is the same as the *Cycle* type, with the exception that it stores *CHTLineSegs* instead of normal line segments.

---

algorithm *build_convex_hull_tree*(*polygon*)
input: A *Cycle polygon*.
output: A *CHTNode* which is the root of the convex hull tree for *polygon*.
method:
  CHTNode cl := ∅ ;
  Cycle ch := the convex hull of *polygon*; let *ch* = {cs_1, ..., cs_n};
  for each $cs_i \in ch$ **do**
    if ($cs_i \notin$ *polygon*) (that is, it was added to make the polygon convex) **then**
      cp := cs_i and the segments in *polygon* which were replaced by cs_i;
      cch := *build_convex_hull_tree*(*cp*)
    else
      cch := ⊥
    end if;
    cl := *cl* ∪ {(cs_i.*u*, cs_i.*v*, *cch*)}
  end for;
  return *cl*
end *build_convex_hull_tree*

Figure 10: Algorithm *build_convex_hull_tree*

Our implementation uses the Graham scan from [G72] to compute the convex hull in $O(n \cdot \log n)$ time for a given polygon with n vertices. This must be performed once for the whole object and once for each concavity. Because

the number of vertices in all the concavities at each level of the tree is less than or equal to n, the total time for computing convex hulls is bounded by $O(dn \cdot \log n)$, where d is the depth of the convex hull tree.

The line segments in the convex hull will be returned in counterclockwise order by the procedure for computing the convex hull. If the line segments in the given polygon are also ordered in this way, discovering which segments from the convex hull are not in the original region and discovering which segments they have replaced can be done in linear time by going through both lists in parallel, and testing for equality. When the two segments are not equal, go through the list from the original polygon and put segments into a separate list L until a segment with end point equal to the end point of the segment from the convex hull is found. List L will then contain the segments which were replaced by the segment in the convex hull. The only problem with this algorithm is finding where in the two lists to start, because the starting segment must be in both sets. This can be done by marking which segments are in the convex hull and which are not during the construction of the convex hull, and then testing the lines in the region beginning with the first until one is found which is on the convex hull. This takes $O(n)$ time. Finding which element of the hull is equal to this segment can then also be done in $O(n)$ time. Marking whether the segments are in the convex hull or not does not change the asymptotic running time of the Graham scan. Because this linear running time is less than the time taken by the Graham scan, the running time of the entire algorithm is equal to the running time of the Graham scan.

Theorem 2: For a given polygon with n vertices, the convex hull tree can be built in $O(dn \cdot \log n)$ time, where d is the depth of the resulting tree.

To recreate the polygon which is represented by a convex hull tree, start with the root node and do the following:

- For each segment in the node which does not have a child, return that segment.
- For each segment in the node which has a child, go to that subnode and use this procedure on that node.

5 Matching Corresponding Components

We now address the problem of matching components in one snapshot with components of the other which comes in three flavors:

- Given observations of a moving region consisting of several faces, which faces in the older snapshot correspond to which faces in the newer one?
- Given a moving face with several holes, which holes in the old snapshot correspond to which holes in the new?
- Given a moving face (cycle) with concavities and two snapshots of it, which concavities in the old and new snapshots correspond to each other?

Figure 11 illustrates the problem. It becomes aggravated by the fact that components may split or merge between snapshots.

In all three cases we need to find matching pairs of cycles (i.e., simple polygons). From now on we assume that two sets of cycles C and D are the given input for this problem.

5.1 Requirements for Matching

Before discussing strategies for matching, we should understand the quality criteria for such strategies.

1. It seems obvious that matching should work correctly for any component that has not moved at all.
2. Components that have moved a small distance relative to their size should be matched correctly.

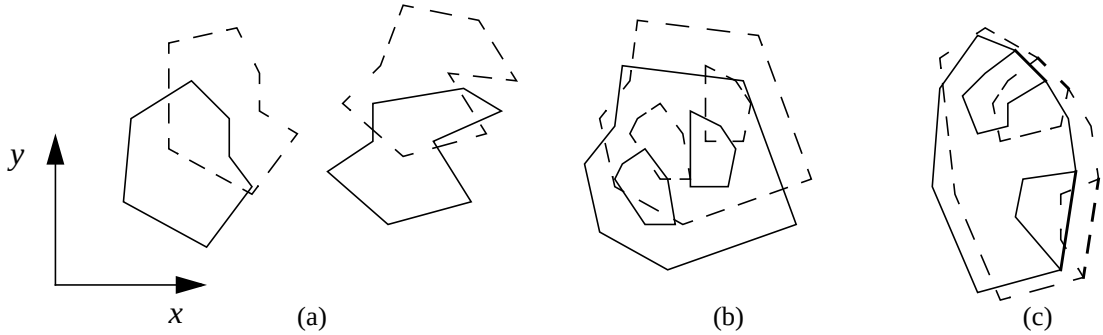


Figure 11: Matching components of moving region observations: (a) faces, (b) holes, (c) concavities

3. It should be possible to match components that have had minor changes to their shape and size.
4. A matching algorithm should discover that a component has been split into fragments or merged from them.
5. A matching strategy should offer criteria to judge the quality of observations. In other words, it should allow one to decide whether two successive snapshots are close enough in time, or too far apart.

Generally, it seems reasonable to require that a matching strategy is guaranteed to produce correct matchings for the components of a moving region if the frequency of observations is increased. This can be formulated a bit more precisely as follows:

Definition 3: Let *mr* be a moving region with several components, and let S_1 and S_2 be two observations of it at times *t* and $t + \Delta t$. A matching strategy is called *safe*, if it is guaranteed to produce a correct matching of the components of *mr* if $\Delta t \to 0$. In other words, there exists an $\varepsilon > 0$ such that the matching is correct for all $\Delta t \leq \varepsilon$.

5.2 Strategies for Matching

Strategies for matching include the following:

1. *Position of centroid*. For each cycle, compute its centroid (center of gravity). This transforms each set of cycles into a set of points. A closest pair in the point sets *C*' and *D*' is a pair of points (*p*, *q*) such that *q* is the point in *D*' closest to *p* and *p* is the point in *C*' closest to *q*. For each closest pair, match the corresponding cycles.
2. *Overlap*. For each pair of cycles *c* in *C* and *d* in *D* compute their intersection area *u* and take the relative overlap, that is, *overlap*(*c*, *d*) = *size*(*u*)/*size*(*d*) and *overlap*(*d*, *c*) = *size*(*u*)/*size*(*c*). The *overlap* relationship can be represented as a weighted directed graph (i.e. if *overlap*(*c*, *d*) = *k*, for *k* > 0, then there is an edge from *c* to *d* of weight *k)*. Then there are several options:
    a. *Fixed threshold*. Introduce a threshold *t* (e.g. *t* = 60%). Two cycles *c* and *d* match if *overlap*(*c*, *d*) > *t* and *overlap*(*d*, *c*) > *t* .
    b. *Maximize overlap*. For all cycles (nodes) order their outgoing edges by weight. For a node *c* let $succ_1(c)$, ..., $succ_n(c)$ be its ordered list of successors. Match *c* with *d* if $d = succ_1(c)$ and $c = succ_1(d)$.

So far we have considered the matching of single cycles. However, the overlap graph allows us to recognize in a natural way transitions where cycles split or merge. See Figure 12. Here *c* splits into *d*, *e*, and *f* (or is a merge of *d*, *e*, and *f*). This can be deduced from the fact that for each of the three fragments the overlap with *c* is large (above

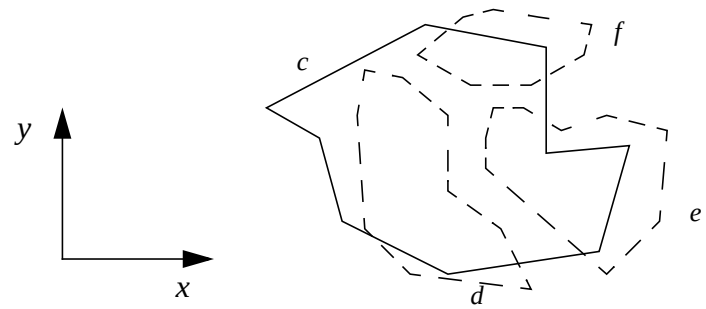


Figure 12: Cycle *c* splits into three cycles *d*, *e*, and *f*.

50 %, say) whereas for *c* the overlap with either *d*, *e*, or *f* is relatively small, but the *sum* of their overlaps is large. This leads to strategies for matching a cycle with a set of cycles:

c. *Fixed threshold, set of cycles.* As in (a), introduce a threshold *t* (e.g. *t* = 60%). Match *c* with $\{ d \in D \mid overlap(c, d) > t\} \cup \{ d \in D \mid overlap(d, c) > t\}$.
d. *Maximize overlap, set of cycles.* Order outgoing edges by weight as in (b). Match *c* with $\{succ_1(c)\} \cup \{ d \in D \mid c = succ_1(d)\}$.

What is a good strategy in the light of the requirements of Section 5.1? Using the centroids, although simple, is not a safe strategy. This is because centroids may lie outside their cycles so that centroids even of disjoint cycles may coincide. This can lead to entirely wrong matchings. The overlap techniques are safe because overlaps approach 100% for region observations when $\Delta t \to 0$. Of course, snapshots have to be close enough to ensure reasonable results.

In the remainder of this paper, we will restrict attention to considering a single cycle with concavities, represented in a convex hull tree. The full paper [TG01] covers the general case with multiple faces and holes. However, the techniques for matching components are already needed in the restricted case for matching concavities in two snapshots of a single cycle. Also, we need to treat transitions such as the splitting/merging of concavities.

5.3 Matching Two Convex Hull Trees

To support the matching of concavities, we compute for two given convex hull trees an overlap graph. Its nodes are the nodes of the convex hull trees; to store the edges, the data structure for nodes is extended to store also a set of pointers to other nodes; each pointer has an associated weight indicating the overlap.

type *OverlapEdge* = { (*node*, *weight*) | *node* ∈ *CHTNode*, *weight* ∈ *real*}
CHTNode **subtype** *CHTNodeWO* = { (..., O) | O ⊂ *OverlapEdge*}

In the description of algorithm *compute_overlap_graph* (Figure 13) we assume that the two argument convex hull trees have been constructed using nodes of type *CHTNodeWO* (“convex hull tree node with overlap”) and that in each node the set *O* of overlap edges has been initialized to the empty set. This is a trivial modification of algorithm *build_convex_hull_tree*.

The algorithm traverses the tree, computing the overlap for pairs of nodes of different trees at the same level whose parents overlap. If the two nodes overlap at a percentage higher than *criterion*, then the nodes are linked.

The intersection of two convex polygons with *l* and *m* edges can be computed in time O(*l* + *m*) (see e.g. [PS85, Theorem 7.3]). If the two polygons represented in the convex hull trees have a total of *n* edges, then the running time for *compute_overlap_graph* can be bounded by $O(d \cdot f^2 \cdot n)$, where *d* is the depth of the tree and *f* the maxi-

algorithm *compute_overlap_graph*(cht_1, cht_2, *criterion*)
input: Two convex hull trees cht_1 and cht_2 with nodes of type *CHTNodeWO* and the real number *criterion*, which controls how much two convex hull tree nodes must overlap to be considered a match.
output: cht_1 and cht_2 are updated to contain overlap edges for matching pairs of nodes.
method:
overlap := *intersection*(cht_1, cht_2); //intersection of convex polygons in the roots
$overlap_1$:= (*area*(*overlap*)/*area*(cht_1))*100;
$overlap_2$:= (*area*(*overlap*)/*area*(cht_2))*100;
if ($overlap_1$ > *criterion*) **and** ($overlap_2$ > *criterion*) **then**
OverlapEdge oe_1 := (cht_2, $overlap_1$);
OverlapEdge oe_2 := (cht_1, $overlap_2$);
$cht_1.O := cht_1.O \cup \{oe_1\}$; $cht_2.O := cht_2.O \cup \{oe_2\}$;
for each son s_1 of cht_1 **do**
for each son s_2 of cht_2 **do**
compute_overlap_graph(s_1, s_2, *criterion*)
end for
end for
end if
end *compute_overlap_graph*

Figure 13: Algorithm *compute_overlap_graph*

mal fanout, since on each level of the tree there are less than n edges and overlap computation is called for each combination of f sons of a node. – Our implementation described in Section 7 uses a function for computing the intersection of two polygons that comes with *java 1.2* (*java.awt.area*) and the authors do not know what algorithm is used there.

6 Interpolating Between Two Arbitrary Polygons

We are now ready to address the problem of interpolating between two general, possibly non-convex polygons. We assume these polygons are represented by convex hull trees for which the overlap graph has been computed.

The basic idea is, of course, to use the *rotating_plane* algorithm from Section 3 on each matching pair of nodes of the two convex hull trees. Let us consider what can happen for a concavity from one snapshot to the next.

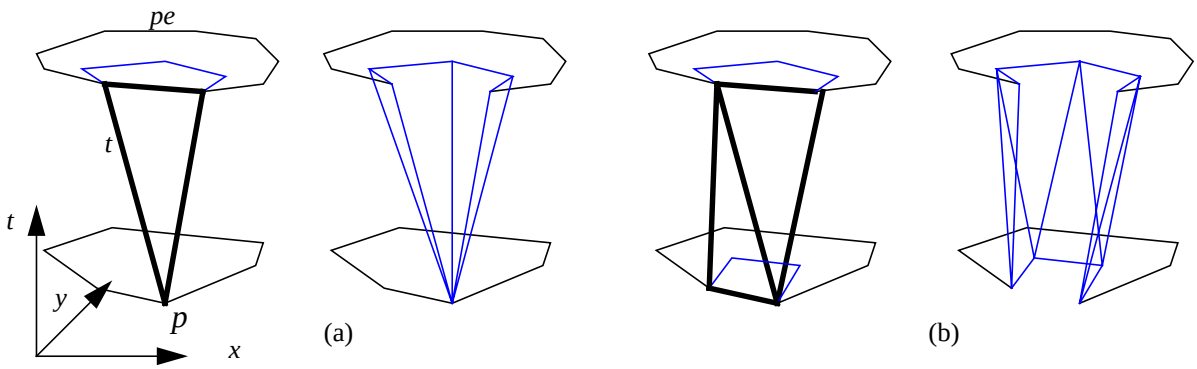


Figure 14: Transitions for cancavities: (a) unmatched concavity, (b) two matching single concavities

The first case (see Figure 14 (a)) is that the concavity doesn't find a "matching" partner in the other polygon. In this case we consider the trapezium *t* involving its parent edge *pe* which is most likely a triangle (drawn fat in Figure 14). All the edges of the concavity are connected by triangles with the point *p* in the other polygon in which triangle *t* ends.[1] So the concavity appears to spring from *p* or to disappear into *p* depending on which snapshot is first in time.

Technically, trapeziums are first constructed for the two convex outer polygons, which includes the creation of *t*. Then, trapeziums (triangles) are constructed for the concavity, including its parent edge, so that *t* is created once more. Then the union is formed of the first set and the second set of trapeziums, *subtracting their intersection*. This leads to the complete removal of trapezium *t*.

The second case (Figure 14 (b)) is that there is a single matching partner for the given concavity in the other polygon. Then trapeziums are constructed recursively for the two concavities. Again, this also yields the trapeziums involving the parent edges of the two concavities so that these can be removed from the result when forming the union with the trapeziums from the next higher level.

The third, most involved case occurs if the concavity matches more than one concavity in the other polygon (Figure 15).

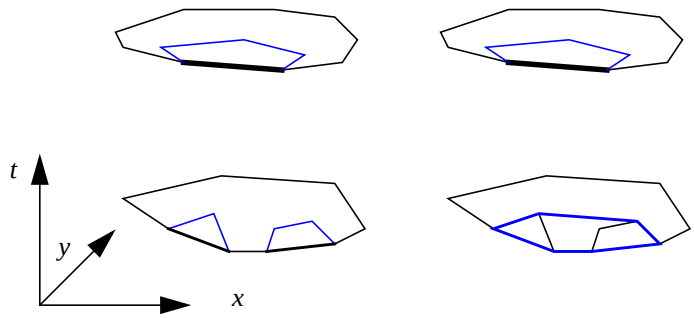


Figure 15: Transitions for concavities: one concavity matches several concavities

In this case, before the interpolation is performed, the set *C* of concavities matching the one concavity is first joined into a single convex polygon. This is done as a transformation on the convex hull tree, which is illustrated in Figure 16.

The algorithm for performing the transformation shown in Figure 16 is called *join_concavities* (Figure 17). It uses a function *recreate_polygon* (Figure 18) implementing the strategy for reconstructing a polygon from a convex hull tree sketched at the end of Section 4. Some of the notations used in *join_concavities* are shown in Figure 16.

Finally, the overall algorithm for interpolating between two polygons is given in Figure 19, Figure 20, and Figure 21. The strategy for matching concavities is actually a mixture of strategies 2c and 2d: The overlap graph is constructed applying a fixed threshold (*criterion*). But then a concavity *c* is matched to all concavities connected by an overlap edge for which it is the maximally overlapping concavity.

The analysis of the complexity of this algorithm is a bit more involved and for lack of space omitted here. In the full paper [TG01] an upper bound of $O(d^2\, n \log n)$ is derived, where *d* is a bound on the depth of the convex hull trees and *n* the total number of edges of both polygons.

1. If *t* is indeed a trapezium which happens if there is a segment *s* parallel to *pe* in the other polygon, then one of the end points of *S* is selected arbitrarily to play the role of *p*.

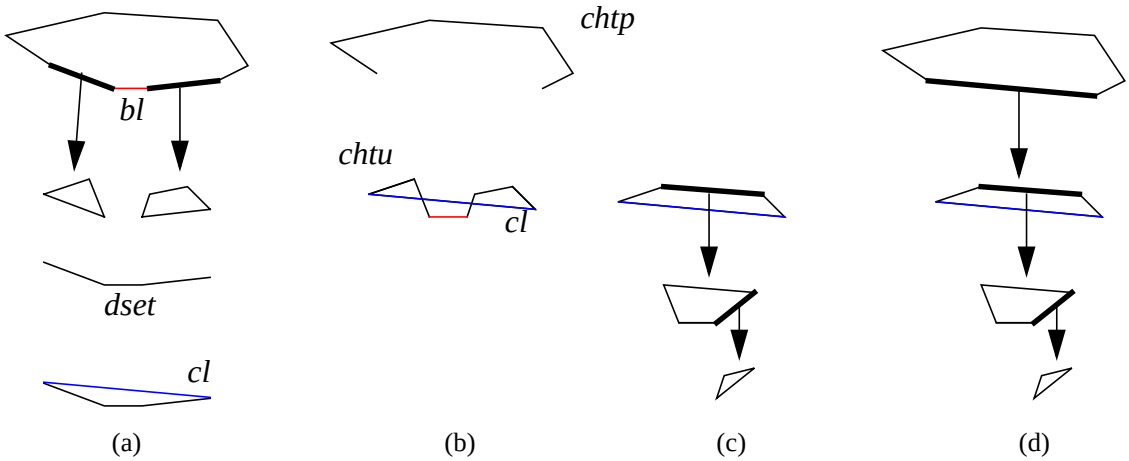


Figure 16: Rebuilding the convex hull tree to join concavities

---

algorithm *join_concavities*(*chts*, *chtp*, *bl*)
input: A set of convex hull tree nodes with overlap graph, *chts*, which represents the concavities to be joined, the convex hull tree node with overlap graph *chtp*, which is the parent node of the nodes in *chts*, and a set of lines, *bl*, which represents the lines between the concavities.
output: A single convex hull tree node which is the union of the others.
method:
let *chtu* be an empty set of line segments;
for each $cht \in chts$ **do**
chtu := *chtu* ∪ *recreate_polygon*(*cht*)
end for;
let *dset* be an empty set of line segments;
dset := *dset* ∪ *bl*;
for each $l \in chtu$ **do**
if $l \in$ *chtp.S* **do**
chtu := *chtu* \ {*l*};
chtp.S := *chtp.S* \ {*l*};
dset := *dset* ∪ *bl*;
end if
end for;
let *cl* be the line segment that needs to be added to *dset* to make it a cycle;[a]
chtu := *chtu* ∪ *bl*;
chtu := *chtu* ∪ {*cl*};
let *res* be the cycle formed by the line segments in *chtu*;[b]
resch = *build_convex_hull_tree*(*res*)
let *rlp* be a *CHTLineSeg* containing the line segment *cl* and a reference to the *CHTNodeWO resch*;
chtp.S := *chtp.S* ∪ {*rlp*};
return *resch*
end *join_concavities*

Figure 17: Algorithm *join_concavities*

a. *dset* now contains all the line segments in the parent that point to the cycles that should be joined. It also contains the lines between them. Because the lines in the parent form a convex polygon, adding only a single line makes this collection of line segments a cycle.

b. Note that the line segments in *chtu* are not necessarily a cycle, because the line *cl* may cross some of the other lines. The implementation contains code that handles this particular case in all functions that normally take cycles as input. The implementation always ignores the line which a node has in common with the parent.

algorithm *recreate_polygon*(*cht*)
input: A convex hull tree, possibly with overlap graph, *cht.*
output: The cycle represented by *cht.*
method:
  let *res* be an empty set of line segments;
  for each *ls* ∈ *cht.S* **do**
    if (*ls* contains link to child node) **then**
      res := *res* ∪ *recreate_polygon*(*ls.child*)
    else
      res := *res* ∪ {*ls*}
    end if
  end for;
  return *res*
end *recreate_polygon*

Figure 18: Algorithm *recreate_polygon*

algorithm *create_moving_cycle*($poly_1$, $poly_2$, t_1, t_2, *criterion*)
input: Two polygons, $poly_1$ and $poly_2$ represented as *Cycles*, two times, t_1 and t_2, representing the times when $poly_1$ and $poly_2$ are valid, and a *criterion* specifying how much overlap is required to consider two objects to match.
output: An *MCycle* resulting from the interpolation of the two polygons.
method:
  cht_1 := *build_convex_hull_tree*($poly_1$);
  cht_2 := *build_convex_hull_tree*($poly_2$);
  compute_overlap_graph(cht_1, cht_2, *criterion*);
  return *trapezium_rep_builder*(cht_1, cht_2, t_1, t_2)
end *create_moving_cycle*

Figure 19: Algorithm *create_moving_cycle*

7 Experimental Results

All algorithms described in this paper have been implemented in Java. The implementation is available on the Web at `http://www.idi.ntnu.no/~tossebro/mcinterpolator/interpolator.html`. There is an applet that allows one to interactively enter two snapshots and then see the interpolation, then a version for download that creates from two snapshots a VRML file which can be studied through a VRML viewer. The documented source code is also available.

The experimental results described next have been derived from this implementation. Matching multiple regions and joining separate regions (discussed in the full paper [TG01]) have not been implemented yet. The current program also has no support for holes which are not concavities. The implementation works for all regions which remain in one piece and do not move much relative to one another. (The more movement or rotation there is, the lower the quality of the resulting triangle representation will be.) The representation created by the algorithm was passed to the extensible database graphical user interface created by Miguel Rodríguez Luaces, which used it to create interpolated values between the two snapshots. All the interpolations shown in this document were created by this program.

An extension which handles multiple regions, regions with holes and regions which split and merge is planned to be built on top of the existing program.

algorithm *trapezium_rep_builder*(cht_1, cht_2, t_1, t_2)
input: Two convex hull trees with overlap graph represented by their roots, cht_1 and cht_2, and two times, t_1 and t_2, when the polygons represented by cht_1 and cht_2 are valid.
output: An *Mcycle* resulting from the interpolation of the two polygons.
method:

```
children1 := the children of cht1; children2 := the children of cht2;
MCycle mc := rotating_plane(cht1, cht2, t1, t2);      // convex hull tree node is a subtype of cycle.
um := children1 ∪ children2;                          // "unmatched children"

// Step 1: Find partners in cht2 for children in children1
for each child ∈ children1 do
    ol := the list of concavities that overlap child (according to the overlap graph);
    // restrict ol to concavities for which child is the maximally overlapping one
    for each c ∈ ol do
        col := the list of concavities that overlap c;
        if not (child is the element of col with greatest overlap) then ol := ol \ {c} end if
    end for;
    if ol ≠ ∅ then
        lsbc := {the line segments that lie between the concavities in ol};
        concavity := join_concavities(ol, cht2, lsbc);
        cr := trapezium_rep_builder(child, concavity, t1, t2)
        mc := (mc ∪ cr) \ (mc ∩ cr);
        um := um \ {child}; um := um \ ol;
    end if
end for;

// Step 2: Find partners in cht1 for yet unmatched children in children2
for each child ∈ (children2 ∩ um) do
    ol := the list of concavities that overlap child (according to the overlap graph);
    for each c ∈ ol do
        col := the list of concavities that overlap c;
        if not (child is the element of col with greatest overlap) then ol := ol \ {c} end if
    end for;
    if ol ≠ ∅ then
        lsbc := {the line segments that lie between the concavities in ol};
        concavity := join_concavities(ol, cht1, lsbc);
        cr := trapezium_rep_builder(child, concavity, t1, t2);
        mc := (mc ∪ cr) \ (mc ∩ cr);
        um := um \ {child}; um := um \ ol;
    end if
end for;
```

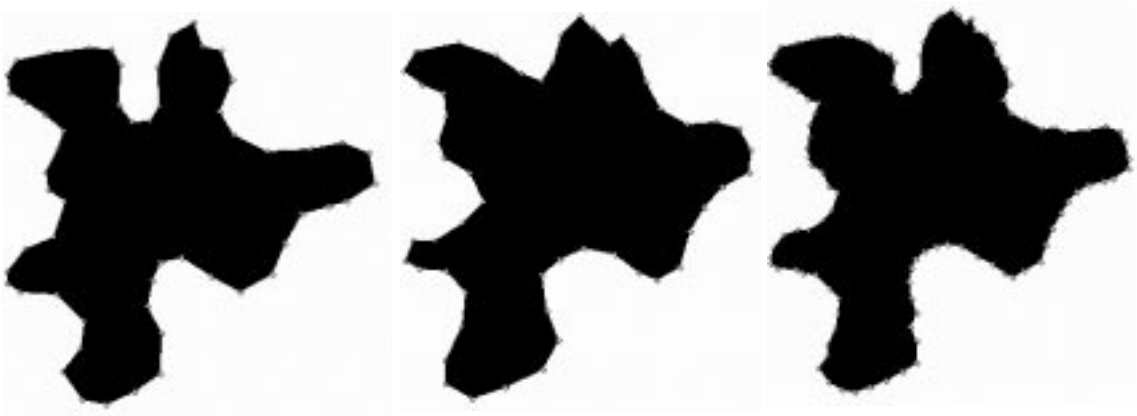
Figure 20: Algorithm *trapezium_rep_builder*, Part 1

For the artificial test cases which were used to test the program for bugs, the results have in most cases become fairly good, such as in Figure 22. However, the algorithm is very sensitive to overlap, and the smaller concavities may be erroneously matched to points if they have moved a large distance relative to the size of the concavity. This problem may be reduced by reducing the threshold overlap, that is, how much should two concavities overlap to be considered to match. The danger of reducing this criterion is that concavities might be matched erroneously if they overlap by a small percentage (The program always matches the object to the first object or combination of objects which match by more than the criterion). The overlap percentage was lowered several times during testing. The first tests were conducted with an 80% overlap requirement, while the last tests used a

```
    // Step 3: Connect still unmatched children with points (Figure 14 (a))
    for each child ∈ ((children_1 ∪ children_2) ∩ um) do
        li := the line in the parent containing the pointer to child;
        ml := the moving line segment in mc which contains li;
        cp := one of the points in ml but not in li;
        for each line segment l in recreate_polygon(child) do
            ms := a moving line segment connecting l as and cp (a triangle);
            mc := mc ∪ {ms}
        end for;
        change ml such that it no longer contains li, but only one end point from li. If
        this turns it into a moving point, remove it entirely
    end for;
    return mc
end trapezium_rep_builder
```

Figure 21: Algorithm *trapezium_rep_builder*, Part 2

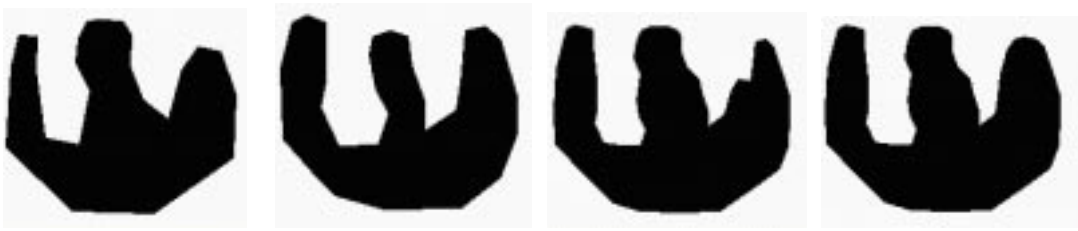
requirement of 10%. 5% may be even better in some cases, but with such a small overlap criterion, there is a danger of matching the concavities wrongly due to small overlaps with other concavities. Note that this problem is more likely to occur for high snapshot distances. With a very small snapshot distance, the concavities have moved little, and overlaps between "non-matching" concavities will be unlikely. With a greater snapshot distance, these overlaps may be significant. Figure 23 shows two interpolations, one with a criterion of 40% and one with a criterion of 10%. The one with 10% clearly looks better than the one with 40%, especially the right part of the figure.



First snapshot Second snapshot Interpolated value

Figure 22: Interpolation of regular object

Another problem which has occurred in a few cases is that the convex hull trees have become slightly different for very similar regions, causing some strange behavior by the matching algorithm. A specific example of this is shown in Figure 24, where changing the position of a single point causes a line which previously belonged to the convex hull of the region to instead belong to the concavity. In this particular example, concavity *a* will be matched to point *b*. When the larger concavities are first matched, the thin line in concavity *a* will be matched to point *b* by the rotating plane algorithm. When concavity *a* is then added, and no matches are found for it, all the lines in it will be matched to point *b*. This interpolation artifact is clearly seen in the interpolations shown in the right part of Figure 24, where the interpolation has two "teeth" instead of the single ending in the two snapshots. To get a good-looking result, the two lines from the real region in concavity *a* would have to be matched to the



First snapshot | Second snapshot | Interpolation, overlap criterion 40%. | Interpolation, overlap criterion 10%.

Figure 23: Test object with original snapshots and interpolated values

two lines *c* and *d* in the other snapshot. The program does not discover this matching because of the different positions of these two lines in the convex hull tree.

However, this problem is most often caused by using objects with few lines and sharp angles, and is therefore less likely to happen with real objects. For instance, in the first example in Figure 24, a real object would probably have a rounded corner, which would have caused a small concavity which might be matched to concavity *a* in the rightmost figure. In the few remaining cases the concavities will likely be so small and/or thin that it will be hard to observe the error. However, for test data which use relatively few lines, this will continue to be a problem.

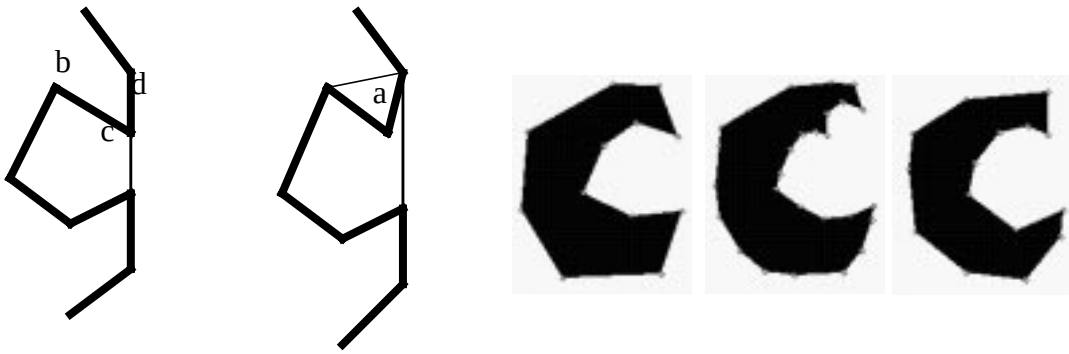


Figure 24: Convex concavity becomes non-convex.

8 Related Work

As mentioned in the introduction, algorithms for creating interpolations between two snapshots already exist. One of these, [SG92], was designed to help creators of animated movies by generating intermediate shapes between two snapshots of cartoon figures. This is a very similar problem to the shape interpolation done by the rotating plane algorithm and convex hull tree in this paper. From the examples they have presented, it seems that their approach is better at preserving shape and avoiding some strange behaviors than ours. It is also definitely better at rotation, which it seems able to detect and account for. The problem, however, is that the user of the system must specify seven constants which are used in the interpolation. They present a table with the numbers they have used in each of their examples, and for four of the constants they are all different. This probably means that there is no set of numbers that works universally. In our approach, one of the goals is that this process should go completely automatically, without any user interaction at all. Also, the preservation of shape is not that important for our application, because the goal is to store a representation for amorphous objects and not objects with a fairly fixed shape, such as the dancing person used as an example in [SG92]. Another problem is the running time. If the user

specifies an initial correspondence between two points, their algorithm runs in $O(n^2)$ time. However, if the user doesn't specify this, it runs in $O(n^2 \log n)$ time. The average case running time of our approach, however, is close to $O(n \log n)$. The d variable is somewhat dependent on n, because more details may be shown. However, it will grow only very slowly.

9 Conclusions

This paper has presented an approach to building the moving region representation described in [FGNS00] from a series of snapshots of an amorphous region. The combination of rotating plane algorithm and overlap graph seems to work well for most regions of this type, although there seem to be better approaches if an interpolation between two snapshots is all that one wants. However, if one instead wants to interpolate between five hundred snapshots, our approach seems to be a good one, because it doesn't demand any user interaction and has a reasonable running time. The algorithms described in the paper have been implemented. The running time has not been a problem with any of the tests that have been run up until now, even though the test program has been implemented in Java. There are some interesting possibilities for future work:

1. The matching strategies described in Section 5 should be implemented and compared systematically. So far we have only implemented one particular choice.
2. A problem with the overlap strategies is that for a large object that is translated in the plane the smaller parts (e.g. lower level concavities) move a lot relatively to their size even though the entire object moves only a little. Hence small concavities will not overlap any more. There are several ways to compensate for this, for example, by combining overlap with a distance criterion for the small components. This should be explored in more detail and evaluated experimentally.
3. Given a large collection of snapshots of an object which moves only little, techniques for data reduction need to be developed. For example, suppose an oil spill in the sea is captured every minute, constructing interpolations between all successive snapshots may lead to an unnecessary amount of data. How can one construct a minimal representation according to some required precision?
4. Precise definitions for the quality of a series of snapshots should be developed. This should allow one to decide whether a series of observations is "good enough". Such definitions could be given in terms of the matching strategies described in the paper.

References

[BGE+00] M. H. Böhlen, R. H. Güting, 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 (2000)*, pp. 1-42.

[CG94] T. S. Cheng and S. K. Gadia, A Pattern Matching Language for Spatio-Temporal Databases. In Proc. ACM Conf. on Information and Knowledge Management, pp. 288-295, 1994.

[CR97] J. Chomicki and P. Revesz, Constraint-Based Interoperability of Spatio-Temporal Databases. In Proc. 5th Int. Symp. on Large Spatial Databases, pp. 142-161, Berlin, Germany, 1997.

[CR99] J. Chomicki and P. Revesz, A Geometric Framework for Specifying Spatiotemporal Objects. In Proc. 6th Int. Workshop on Temporal Representation and Reasoning (TIME), pp. 41-46, 1999.

[FGNS00] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider, A Data Model and Data Structures for Moving Objects Databases. Proc. ACM SIGMOD Int. Conf. on Management of Data, Dallas, Texas, pp. 319-330, 2000.

[G72] R. L. Graham, An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters 1 (1972*), pp. 132-133.

[PD95] D. J. Peuquet and N. Duan, An Event-Based Spatiotemporal Data Model (ESTDM) for Temporal Analysis of Geographical Data. *Int. Journal of Geographical Information Systems 9:1 (1995)*, pp. 7-24.

[PS85] F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction. Springer-Verlag, New York, 1985.

[SG92] T. W. Sederberg and E. Greenwood: A Physically Based Approach to 2-D Shape Blending. *Computer Graphics (Proc. ACM SIGGRAPH) 26:2 (1992)*, pp 25-34.

[SWCD97] A. P. Sistla, O. Wolfson, S. Chamberlain and S. Dao: Modeling and Querying Moving Objects. Proc. Int. Conf. on Data Engineering, pp. 422-432, 1997.

[TG01] E. Tøssebro and R. H. Güting, Creating Representations for Continuously Moving Regions from Observations. FernUniversität Hagen, Informatik-Report, in preparation, 2001.

[W94] M. F. Worboys, A Unified Model for Spatial and Temporal Information. *The Computer Journal 37:1 (1994)*, pp. 25-34.