Generating Moving Regions from Snapshots of Complex Regions

MARK MCKENNNEY and ROGER FRYE, Southern Illinois University Edwardsville

Moving regions are a form of spatiotemporal data in which a region changes in shape and/or position over time. In many fields, moving regions representing real-world phenomena are collected using sensors that take temporally encoded snapshots of regions. We provide a novel algorithm that creates a moving region between any two complex regions. The proposed algorithm has worst-case time bounds of $O(n^2)$, but can use approximation techniques to achieve $O(n \lg n)$ in practice, space bounds of $O(n)$, and output size bounded by $O(n)$ (where n is the number of line segments that define the boundaries of the regions).

Categories and Subject Descriptors: H.2.8 [**Database Applications**]: Spatial Databases and GIS; I.3.5 [**Computational Geometry and Object Modeling**]: Geometric Algorithms, Languages, and Systems

General Terms: Algorithms

Additional Key Words and Phrases: Moving regions, region interpolation, polygon morphing

ACM Reference Format:
Mark McKenney and Roger Frye. 2015. Generating moving regions from snapshots of complex regions. ACM Trans. Spatial Algorithms Syst. 1, 1, Article 4 (July 2015), 30 pages.
DOI: http://dx.doi.org/10.1145/2774220

1. INTRODUCTION

The advent of widespread utilization of spatial data for applications ranging from personal use to business and intelligence analysis has cemented the importance of spatial data management as an influence on people's daily lives. Spatiotemporal data combines the spatial aspect of data with a temporal component, reflecting anything from a single timestamp to time ranges that correspond to changing spatial or thematic values over those ranges. In the literature, the term *moving objects databases* was coined to specifically refer to databases that manage instances of the traditional spatial types of *points*, *lines*, and *regions* that change in shape and position over time. For example, a car traveling on a road can be modeled as a moving point that changes position over time, the path of a river that changes over a geologic time frame can be represented as a moving line, and a region representing high levels of sales for a particular product that expands and shrinks over time can be modeled as a moving region. Moving object systems promise the ability to aid in decision support and analysis involving complex data objects that combine spatial, temporal, and thematic data. Furthermore, extensive work on spatiotemporal types, operations, and predicates has paved the way for integration of analysis functionality with existing data systems (Section 2).

A significant challenge to the development of moving object systems is that creating moving object data is nontrivial. Collecting data in the real world often requires the

Authors' addresses: M. McKenney, Department of Computer Science, Southern Illinois University Edwardsville, Edwardsville, IL 62026; email: marmcke@siue.edu; R. Frye, Department of Computer Science, Southern Illinois University Edwardsville, Edwardsville, IL 62026; email: rfrye@siue.edu.

DOI: http://dx.doi.org/10.1145/2774220

use of sensors to generate data representing some aspect of the world. Typically, these sensors collect data at discrete time points, resulting in data *snapshots* at particular instants in time. For example, radar, satellite imaging, and even video formats capture data at discrete time points, despite possibly constructing animations of the data. Each of these snapshots can be associated with their corresponding temporal values, but they remain a discrete sampling of a continuous phenomenon. By sampling the continuous phenomenon, data is naturally lost; reconstructing or approximating the lost data is nontrivial. In general, to reconstruct a moving object from a series of snapshots requires some form of interpolation. General methods of interpolating the trajectory of a moving point from snapshots are straightforward, but interpolating the trajectory of a region poses many difficulties. For instance, a user may want to interpolate the trajectory of a region representing a rain cloud between two snapshots to discover if one's home received rain. In more specific terms, the user wants to create a moving region from snapshots, and determine if that moving region intersects a point at any time. In this situation, two problems turn up: (i) rain clouds often contain multiple, disconnected parts; and (ii) rain clouds sometimes contain holes in their interior where no rain is falling. It is not immediately clear how disconnected components should be interpolated, especially if there are varying numbers of them in each snapshot. A simple visualization of movement may be sufficient for one user to examine one rain cloud, but in a database setting in which a table of rain clouds may be joined with a table of counties using some spatiotemporal predicate, strict data type definitions must be enforced to ensure data integrity and operational consistency. Currently, no algorithms exist that can generate a *valid* moving region between two region snapshots of arbitrary complexity that correspond to accepted spatial region type definitions [Egenhofer and Herring 1990; Güting et al. 2000; Herring 2010]. We use the term *valid moving region* to emphasize that the result of our algorithm is a spatiotemporal object that satisfies the type definition of moving regions (Section 2). An *invalid moving region*, in contrast, is a geometry that does not satisfy the type definition of moving regions (i.e., a moving region that contains a self-intersecting boundary at some time instant).

We denote the problem of creating a valid moving region from two temporally encoded region snapshots as the *region interpolation problem* (RIP). In this article, we provide the first solution to the RIP, in the form of an algorithm, with the following properties:

(1) The algorithm returns a valid moving region for valid input of an arbitrary shape.
(2) The algorithm provides linear bounds on the size of the output with respect to the input.
(3) The algorithm handles complex regions, possibly containing multiple faces and holes.

We consider regions to be *complex regions* (Section 2) corresponding to the type definition in Schneider and Behr [2006], and with boundaries stored as polygonal curves [Lema et al. 2003].

In order to achieve the three properties listed earlier in our solution, we make use of the observation that definitive information on the extent and position of a moving region represented as a series of snapshots is available only at the time instants for which a snapshot exists. Thus, the extent and position of a complex region in between snapshots must be an approximation of the region at those instants. This observation provides flexibility in the interpolation mechanism that we use to generate a moving region from snapshots; in particular, we develop an algorithm that returns a complex region approximating the input regions at all instants in between snapshots. These approximations may be simplified versions of the snapshots or may have geometries that closely align with the structure of the snapshots. However, there are an infinite

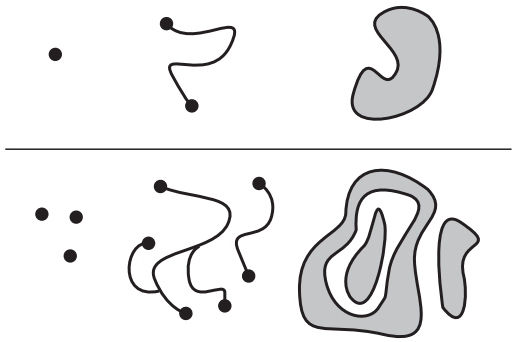


Fig. 1. Example of a simple (top row) and complex (bottom row) point, line, and region.

number of possible interpolations between snapshots, and our algorithm will provide an approximation of them that is guaranteed to satisfy type constraints.

The article is structured as follows. Related work is presented in Section 2, followed by a discussion of required data models and notation in Section 3. We offer an overview of the rest of the article in Section 4. Our solution to the RIP is then developed in stages, beginning with an RIP solution for simple, convex regions in Section 5. Nonconvex regions are handled in Section 6. The solution is extended to handle complex regions containing multiple faces and holes in Section 7. Section 8 presents experimental results showing the running time behavior of the algorithm on various datasets. We present our conclusions in Section 9.

2. RELATED WORK

The classic spatial data types have been defined as points, lines, and regions, either in simple [Egenhofer and Herring 1990; Frank and Kuhn 1986; Frank et al. 1990] or complex [Schneider and Behr 2006] form. In this article, we assume spatial types are embedded in the metric space $\mathbb{R}^2$. A simple point is a single point embedded in space. A simple line is a single, nonself-intersecting line with two end points. A simple region is defined by a single boundary that separates the region's interior from its exterior, and that has a single *face* and no *holes* (i.e., its interior is connected). A complex point contains zero or more simple points. A complex line contains zero or more nonintersecting lines that may have multiple end points (for instance, a line that branches). A complex region can have multiple faces, each possibly containing holes (e.g., Italy with its islands forming faces and a hole where Vatican City lies). Figure 1 depicts representative examples.

The recognition that many spatial objects in reality contain a temporal component led to the development of *moving object* types [Worboys 1994; Sistla et al. 1997; Güting et al. 2000; Lema et al. 2003; Trajcevski et al. 2004; Wolfson et al. 1999]. In Güting et al. [2000], moving counterparts to the traditional spatial types are defined such that a moving object is a mapping from time to a traditional spatial object type. Time is modeled as the set of real numbers. Let α be a traditional spatial type, the moving type of α is then $M_\alpha : \mathbb{R} \rightarrow \alpha$. Some restrictions on moving types are imposed, for example, moving objects must exist and move somewhat continuously. A key restriction, that we denote the *validity constraint*, states that at any given time instant for which a moving object is defined, the temporally static object defined by the moving object at that instant must be a valid spatial object. Essentially, moving objects in this conception are three-dimensional objects in which time forms the third dimension.

Lema et al. [2003] extend the work on moving objects to a discrete representation termed the *slice representation*. The slice representation represents moving objects as

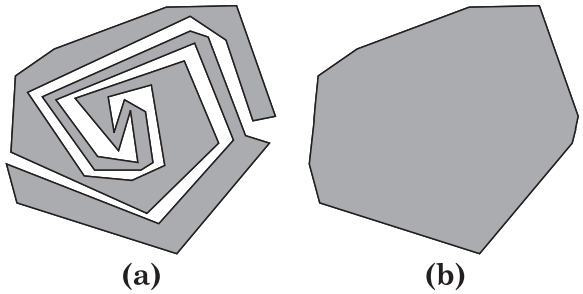


Fig. 2. A region that shows that the assumptions made for the algorithm in Tossebro and Guting [2001] do not always hold. (b) is the convex hull of (a).

a series of *interval regions* such that an interval region represents the movement of a moving object over a time interval between two instants. A moving object is then constructed as a series of interval objects. In essence, the slice representation defines a series of interval object boundaries (snapshots of the moving objects) and the movement of the objects between the snapshots. This concept relates directly to our work, in which snapshots of moving objects, in this case regions, are known and we must compute valid motion across the intervals that satisfy the validity constraint.

In this article, we focus on creating moving regions from region snapshots, a problem traditionally known as *morphing* in computer graphics. Gotsman and Surazhsky [2001], Alexa et al. [2000], Carmel and Cohen-Or [1998], Sederberg et al. [1993], Sederberg and Greenwood [1992], Shapira and Rappoport [1995], and Yang and Feng [2009] investigate morphing between polygons, which correspond to simple regions, or collections of simple polygons. These solutions generally do not apply to the RIP for two reasons: (i) the RIP, as defined, requires interpolations between complex regions that may consist of sets of polygons; and (ii) the semantics of regions as defined for the purposes of this article require strict type definition constraints that are not necessarily enforced by approaches in the literature. In general, the cited approaches are directed towards creating animations between polygonal figures. More complex objects are considered in Yang et al. [2012] and Barbic et al. [2012], but these approaches are designed for the interactive deformation of complex figures, not the generation of movement between snapshots. Polygonal mesh morphing, in which a mesh of simplices (usually triangles) are morphed into a destination mesh, form the basis of many animation techniques. However, source and destination meshes must be homeomorphic [Lee et al. 1999; Alexa 2002; Alliez et al. 2008; van Kaick et al. 2011]. Furthermore, morphing in animation domains tends to be constrained by aesthetics rather than structural type constraints, as required in the RIP. The RIP as stated in this article is geared towards database applications in which data size, data type constraints, and operational closure are of paramount importance.

The closest work to ours appears in Tossebro and Guting [2001], in which a solution to the RIP is proposed. The algorithm in Tossebro and Guting [2001] creates an interval region between two region snapshots, which is our stated goal; however, the proposed algorithm does not return a valid result for all input cases. In other words, valid input exists in which the algorithm produces an interval region that will result in a region with a self-intersecting boundary at some time instants. Their algorithm assumes that a segment vertex cannot change its direction of motion over an interval and that only one interval region is required to create a valid moving region between two input region snapshots; however, Figure 2 provides a case in which these assumptions do not hold. Assume that one must create an interval region between the region shown in Figure 2

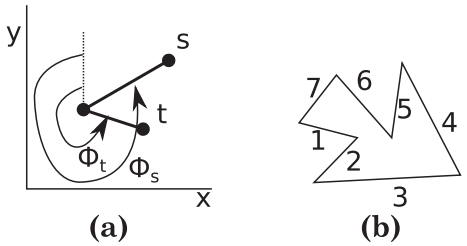


Fig. 3. (a) Two segments, s and t, such that $t < s$. (b) A cycle with segments ordered in cyclic order based on the least segment.

and its convex hull. The points at the end of either of the spirals must travel to a point on the convex hull over the interval. If a vertex cannot change direction over an interval, then no solution exists in which either of those vertices does not cross a boundary segment from the opposing concavity; thus, the assumptions in Tossebro and Guting [2001] do not hold true in general. Note that the objective for Tossebro is to attempt to find a solution that is somewhat similar to the input regions, meaning that the result of that algorithm, if it satisfies spatial type definitions, is likely to more closely approximate the input regions' geometries at all time instants over the interval. Our approach does not attempt to approximate the input geometries as closely, instead focusing on guarantees of correctness. Again, we take advantage of the fact that an interpolation must, using the most basic requirements, simply be an approximation of the input regions at all time instants over an interval.

A solution to the RIP for simple regions appears in McKenney and Webb [2010]. This work extends the work in that article by providing a more detailed solution to the RIP, proving algorithm correctness, and providing a solution to the RIP for complex regions. The solution to the RIP for simple regions is reviewed and expanded in Section 5.

3. DEFINITIONS AND NOTATION

In order to construct an algorithm to address the RIP, we require an *implementation data model* and corresponding notation to represent simple regions, complex regions, and moving regions in a fashion that is suitable for implementing in computer systems but that corresponds to the respective type definitions. Historically, region types have been approximated in computer systems by representing region boundaries as polygonal curves (i.e., collections of straight-line segments); we follow a similar approach. Because a simple region is a special case of a complex region, we use the term "region" to refer to the more general case of complex regions and will explicitly refer to "simple regions" when necessary.

For implementation purposes, the type of *point* is defined as the set of all pairs of real numbers as approximated by floating point numbers, denoted $\mathbb{R}_{fp}$. Let the type $point := \mathbb{R}_{fp} \times \mathbb{R}_{fp}$. Let $p = (x, y) \in point$; p_x refers to x and p_y refers to y. Similarly, a point in three dimensions is $point_{3d} := \mathbb{R}_{fp} \times \mathbb{R}_{fp} \times \mathbb{R}_{fp}$.

The type of *segment* is 2-tuple of pairs of points (p, q) such that $p < q$: $segment := point \times point | (\forall s = (p, q) \in segment : p_x < q_x \vee (p_x = q_x \wedge p_y < q_y))$. For $s = (p, q)$, let $s_p = p, s_q = q, s_{px} = p_x, s_{py} = p_y$, on so on. Let $s \in segment$ and v be an imaginary ray extending vertically from s_p; ϕ_s is the angle covered by rotating v in a counterclockwise direction around s_p until it is collinear with and overlapping s (Figure 3(a)). A total ordering exists over segments; let $s, u \in segment$:

$$s < u \Leftrightarrow \begin{pmatrix} s_p < u_p \vee \\ (s_p = u_p \wedge \phi_s < \phi_u) \vee \\ (s_p = u_p \wedge \phi_s = \phi_u \wedge s_q < u_q) \end{pmatrix} \quad (1)$$

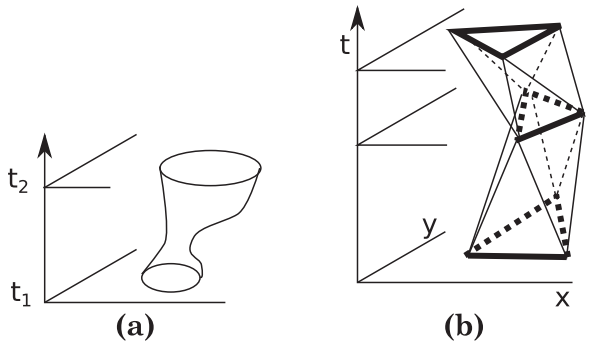


Fig. 4. (a) A moving region as defined in Güting et al. [2000]. (b) A moving region as represented by in an implementation model. (b) contains two interval regions, and depicts the moving segments as delta triangles. Interval region boundaries in (b) are weighted heavier. Line segments hidden from view by the moving region are dashed.

A cycle is a set of segments that form a closed loop such that each segment respectively shares each end point with exactly one other segment, and any pair of segments will intersect either at an end point of both segments or not at all. Let C be a cycle and $G = (V, E)$ be a graph with the end points of segments in C forming the vertices V and segments in C forming the edges E; G must be a connected graph forming a simple cycle:

$$\begin{aligned} cycle := \; & \{C \in 2^{segment} | \; for \; Cit \; holds \; that: \\ & G = (\{p, q | (p, q) \in C\}, \{(p, q) \in C\}) \; is \; connected \; and \\ & forms \; a \; simple \; cycle\} \end{aligned} \tag{2}$$

It is sometimes useful to traverse the segments that form a cycle in the order in which they appear around the cycle. We define a *cyclic order* of line segments forming a cycle to be the order that segments in a cycle are visited if a person walks along the boundary of the cycle beginning at the least most end point of the least most segment towards the greater end point of the least most segment, and continues around the cycle (i.e., counterclockwise around the cycle from the least most segment; Figure 3(b)). When traversing a segment in cyclic order, one end point of a segment will be encountered before the other end point. We refer to the first end point encountered on a segment when traversing a cycle boundary in cyclic order as the *primary point*, and the other as the *secondary point*.

The type of simple region is equivalent to the type of cycle: *simple region* = *cycle*. A complex region is defined by union of one or more cycles, with restrictions formally defined in Schneider and Behr [2006]. Informally, the cycles in a complex region form faces and holes. The boundaries of faces and holes may only intersect at a finite number of points. No cycle can cause another cycle's interior to become disconnected. The boundary of each cycle must separate the interior of the region from its exterior.

A moving region is defined as a mapping M from time instants to complex regions [Güting et al. 2000] where τ is the set of all time instants: $M : \tau \rightarrow complex \; region$ (Figure 4(a)). Therefore, at any given time instant, a valid region exists (even if that is the empty region). Just as regions are defined by their boundary line segments, moving regions will be defined by their boundary line segments that change in shape and position over time. In other words, the boundary of a moving region will be a set of *moving segments* [Lema et al. 2003]. A moving segment defines the motion of a segment across a time interval from t to t'. For implementation purposes, we assume that a moving segment will be defined as a point at one time instant and a segment at the opposing time instant; therefore, a segment will either contract to a point over the time

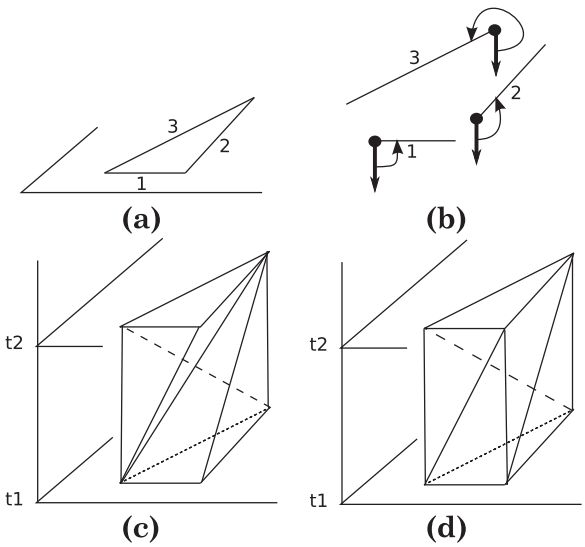


Fig. 5. Two solutions to the RIP (c and d) from identical source and destination regions (a). Segments in the source and destination region are labeled and the progress angles are shown (b). Dotted lines are interval boundary region segments that are hidden from view. Dashed lines are delta triangle edges that are hidden from view.

interval or a point will expand into a segment over the time interval. In essence, the end points of the segment and the point at the opposing time instant, when plotted in three dimensions with time as the third dimension, form a triangle. We therefore represent a moving segment as 3 three-dimensional point coordinates such that the least most point in the segment is always recorded first, the other segment end point is recorded second, and the point at the opposing time instant is recorded third (Figure 4(b)). The type of moving segments is then:

$$\begin{aligned} &\textit{moving segment} := \textit{point}_{3d} \times \textit{point}_{3d} \times \textit{point}_{3d} \textit{ such that :} \\ &\forall (p, q, u) \in \textit{moving segment} : ((p_x, p_y), (q_x, q_y)) \in \textit{segment} \\ &\wedge (u_x, u_y) \in \textit{point} \wedge p_z = q_z \wedge p_z \neq u_z \end{aligned} \tag{3}$$

We use the term *delta triangle* to refer to the triangle formed by plotting the points defining a moving segment in three-dimensional space with time as the third dimension. Note that a moving segment is defined only as a set of three-dimensional points. The implication of this is that the motion of the end points of the moving segment is constant across a time interval, as no motion function is given explicitly (Figure 4(b)).

An *interval region* [Lema et al. 2003] is a collection of moving segments that describe the motion of a region over the time interval defined by two instants. An interval region is a set of moving segments such that the segments at each boundary form valid regions. The interiors of all delta triangles imposed by an interval region must be disjoint. Furthermore, an interval region must satisfy the semantic constraints imposed on moving regions, namely, that the interval region must describe a valid complex region at every time instant covered by the interval. Let $\textit{extract} : \mathbb{R}_{fp} \times \textit{interval region} \rightarrow \textit{complex region}$ be an operation that computes the complex region represented by a set of moving segments at a time instant. We denote the term *interval boundary regions* to describe the two regions defined by an interval region at either end of the interval. The interval boundary region at the earlier time is called the *source region* and the region at the later time is called the *destination region*. For example, Figure 5 shows a source region, destination region, and interval region, including delta triangles.

4. OVERVIEW

In the remainder of this article, we incrementally define a general solution to the RIP for complex regions by defining the RIP for a series of subproblems of increasing complexity. Specifically, we consider the creation of an interval region from two region snapshots, meaning that the general approach is to construct a set of moving segments that fit the definition of an interval region between two temporally referenced region snapshots. A user can construct a complete moving region from a set of such interval regions. We first consider the RIP in the context of convex simple regions, then expand it to handle nonconvex simple regions. The solution to the RIP for complex regions is achieved by interpreting a complex region in such a way that the RIP algorithm can then treat it as if it were a simple region. We show the correctness of each step before progressing to the next. Recall that a significant property of our solution to the RIP is that it always returns a valid interval region for valid region input, providing type consistency that is crucial in database applications.

Note that our strategy to define a solution to the RIP is to create a solution to the RIP for simple regions, then provide algorithms to add structures to complex regions such that they may be interpreted as simple regions and used as input to the algorithm developed for simple regions. Therefore, Section 5 and Section 6 deal explicitly with simple regions. The algorithms provided in those sections are shown to be correct. Once the algorithms for simple regions are established, Section 7 provides algorithms that compute bookkeeping structures that allow complex regions to be interpreted as simple regions and used as input to the algorithms in Section 5 and Section 6. The algorithms in Section 5 and Section 6 are constructed with the foresight that complex regions will eventually be handled.

5. CONVEX SIMPLE REGIONS

Let S and D be two convex simple regions representing snapshots of some moving region observed at times t_0 and t_1, respectively, such that $t_0 < t_1$. A solution to the RIP for S and D will be an interval region I with source region S and destination region D. Because interval regions are defined as sets of moving segments, the solution to the RIP consists of creating two mappings, one from each segment in S to a point in D (i.e., $f_1 : S \rightarrow \{p, q | (p, q) \in D\}$) and one from each each point in S to a segment in D (i.e., $f_2 : \{p, q | (p, q) \in S\} \rightarrow D$) such that the semantic constraints of interval regions are satisfied. Mappings from segments to points and points to segments are required since moving segments are defined as segments that converge to a point over a time interval, and vice versa. The semantic constraints of an interval region require that a nonmoving region extracted at any instant from an interval region must be a valid region. For an interval region with source and destination regions consisting of convex simple regions, an alternative way of stating the constraint is that the delta triangles defined by an interval region plotted in three-dimensional space, along with the source and destination regions, form a polyhedron that is topologically equivalent to a cylinder. Therefore, a region extracted at any time within the interval will result in a slice of the polyhedron parallel to the spatial dimensions, and that slice will be a simple region (although not necessarily convex without further constraints on the interval regions).

Multiple valid interval regions are possible between a source and destination region, as depicted in Figure 5(c) and Figure 5(d). A general algorithm to construct such a mapping must avoid two specific pitfalls. First, each delta triangle corresponding to a moving segment in the resulting interval region must share two edges with adjacent triangles; in other words, the geometry resulting from plotting the computed moving segments, source, and destination regions must not have gaps (recall that it must

form a polyhedron). Second, no two delta triangles may intersect in their interiors; such a situation would result in time instants in which a region extracted from the interval region would have a self-intersecting boundary. Because we are considering convex simple regions, the only situation in which intersecting delta triangles can occur is if the plotted interval region is nonconvex, or in the case of *twisting*, when the resulting interval region has a much smaller volume towards its temporal center than it does at the edges. Twisting can occur from creating interval regions from source and destination regions with different numbers of boundary segments, or with widely variable lengths of boundary segments, or from an algorithm that achieves a poor choice of mappings when creating moving segments. Thus, any algorithm must avoid extreme twisting.

To ensure that an interval region contains no gaps, we construct the mappings of segments to points and points to segments in cyclic order. This straightforward approach ensures that moving segments will share end points with two adjacent moving segments at all times across the time interval. The algorithm will begin by finding the least segments s and d, respectively, in source region R and destination region D. The first step will be to create a moving segment from s to the primary end point of d and a second moving segment from d to the secondary end point of s. At this point, both segments have been used to create moving segments, so we must *increment* one of them by assigning it the next segment in cyclic order from itself in its respective region. Thus, either s will be assigned the next segment in cyclic order from R, or d will be assigned the next segment in cyclic order from D. We call this step the *increment step*, and define its functionality more specifically later. Once one segment has been incremented, the next iteration of the algorithm will always be considering a segment that has been used to create a moving segment, and a segment that has not been used as such. Therefore, each iteration of the algorithm will create a moving segment from the segment that has not yet been used as such to the secondary point of the opposing segment. Then, the increment step will take place, and the algorithm will proceed in its next iteration. The algorithm halts once all segments from both regions have been used to create moving segments.

Discouraging twisting of an interval region is enforced by the choice of the *increment step*, and is achieved by making the observation that although a single segment on one temporal boundary of an interval region will only map to a single point on the opposing temporal boundary, a point on a temporal boundary of an interval region may map to multiple adjacent segments on the opposing temporal boundary. We introduce a metric called the *progress angle* to determine which segment is *incremented* in the increment step of each iteration of the algorithm. Let s be a segment; we define the progress angle for s, called θ_s, to be the angle formed by rotating an imaginary ray shooting in the negative y direction counterclockwise around the primary end point of s until it is collinear and overlapping with s (Figure 5(b)). Furthermore, we use the convention that a progress angle of 0 degrees is interpreted as a progress angle of 360 degrees. The increment step proceeds simply by incrementing, in cyclic order, the segment with the least progress angle. Algorithm 1 specifies the complete algorithm. Lines 18 through 23 create a moving segment from whichever segment has not been used in a moving segment thus far to the secondary point of the opposing segment. Lines 9 through 16 handle the special case in which a moving segment is created with a primary end point in the first iteration of the loop. Lines 25 through 33 perform the increment step. Figure 5(d) depicts the output of Algorithm 1 for source and destination regions equivalent to that shown in Figure 5(a).

LEMMA 5.1. *Algorithm 1 produces a valid interval region from valid, temporally encoded, temporally disjoint source and destination regions. Furthermore, the source*

ALGORITHM 1: An algorithm to Create an Interval Region from Convex Simple Regions. MIN and MAX (Lines 1 through 4), respectively, return the least and greatest segments from the argument region.

```
   Input: A source region R and a destination region D
   Output: An interval region I represented as a set of moving segments
 1 var r ← MIN(R);
 2 var d ← MIN(D);
 3 var r_L ← MAX(R);
 4 var d_L ← MAX(D);
 5 θ_r ← the progress angle of r;
 6 θ_d ← the progress angle of d;
 7 var r_prev, d_prev ← an empty segment;
 8 repeat
 9     if d ≠ d_prev ∧ r ≠ r_prev then
10         if θ_r < θ_d then
11             create moving segment that travels from r to d.primaryPoint and add it to I;
12             create moving segment that travels from d to r.secondaryPoint and add it to I;
13         else
14             create moving segment that travels from d to r.primaryPoint and add it to I;
15             create moving segment that travels from r to d.secondaryPoint and add it to I;
16         end
17     else
18         if d ≠ d_prev then
19             create moving segment that travels from d to r.secondaryPoint and add it to I;
20         end
21         if r ≠ r_prev then
22             create moving segment that travels from r to d.secondaryPoint and add it to I;
23         end
24     end
25     r_prev ← r;
26     d_prev ← d;
27     θ_r ← the progress angle of the segment after r in cyclic order from R;
28     θ_d ← the progress angle of the segment after d in cyclic order from D;
29     if r ≠ r_L ∧ (θ_r < θ_d ∨ d = d_L) then
30         r ← the segment after r in cyclic order from R;
31     else if d ≠ d_L then
32         d ← the segment after d in cyclic order from D;
33     end
34 until (r = r_L ∧ r = r_prev) ∧ (d = d_L ∧ d = d_prev);
```

region, destination region, and delta triangles of the resulting interval region form a convex polyhedron.

PROOF. The algorithm proceeds by essentially *wrapping* delta triangles around the source and destination region to form a polyhedron. One can imagine wrapping paper around two convex polygons to create a tube, albeit an irregular tube.

The first step in the algorithm is to create two moving segments between the two segments with minimal progress angles from the respective source and destination regions (Lines 9 through 16). A moving segment m forms a delta triangle that defines a plane. We use the notation m_p to identify the plane defined by moving segment m. Let the progress angle of plane m_p be identical to the progress angle of m. Thus, it follows that the progress angles of the first two segments are ascending, as are the progress angles of their associated planes, and that the corresponding delta triangles share an

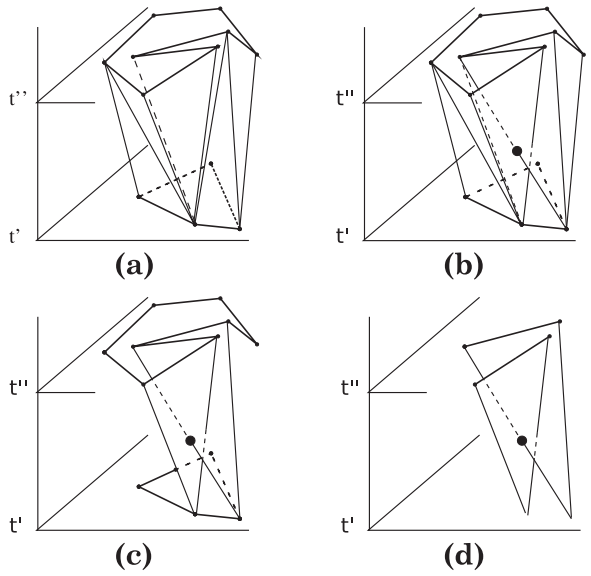


Fig. 6. Possible interval regions from a region at time t' to a region at time t''. Figure (a) shows a valid interval region. Figure (b) shows an invalid interval region. Intersecting delta triangles are isolated in (c) and (d).

edge connecting one end point from each input region. In each successive step, a new moving segment is created such that its delta triangle shares an edge with the delta triangle of the previously created moving segment such that the edge connects an end point from each input region (Lines 18 through 23). The fact that the new delta triangle shares an edge with the previous one is guaranteed by Lines 29 through 33, in which moving segments are formed in cyclic order. Therefore, no *gaps* will appear in the side walls of the resulting structure, and a polyhedron will be formed. In order for us to guarantee that a convex polyhedron is formed, we must ensure that the progress angle of the plane defined by the delta triangle of each successively created moving segment is monotonically increasing, as is enforced by the choice of which segments to advance in Lines 25 through 33. Because the delta triangle of each newly generated moving segment shares an edge with the previously generated delta triangle, and the progress angle of the plane associated with the new delta triangle is greater than that of the plane associated with the previous delta triangle, we will never have an decrease of the progress angle. The resulting polyhedron will be a convex polyhedron. □

6. NONCONVEX SIMPLE REGIONS

Convex regions have the property that when traversing the boundary of a convex region in cyclic order beginning with the least most segment, the progress angle of each segment increases monotonically. Algorithm 1 takes advantage of that property to ensure valid interval regions. Nonconvex regions do not exhibit such a property for progress angles, thus the increment step used in Algorithm 1 does not apply effectively to nonconvex regions. In nonconvex regions, progress angles decrease when traversing segments that form a concavity in the region. The definition of simple regions allows for concavities with a highly complex structure, creating a challenge. For example, Figure 6 depicts two possible sets of moving segments between a source and destination region: (a) a set forming a valid interval region, and (b) a set forming an invalid interval region due to the generation of regions at instants with self-intersecting boundaries. Although

the regions in Figure 6 have a relatively simple structure, one can easily imagine more complex concavity structures that exacerbate the problem.

6.1. Using Convex Hulls to Alleviate Problems Caused by Concavities

Although concavity structures prevent the direct use of a progress angle-based increment operation, as in Algorithm 1, a relatively straightforward solution exists that allows us to preserve the core functionality of Algorithm 1. Our solution is to compute the convex hull of a nonconvex boundary region. Each concavity of a complex region forms a nonself-intersecting polygonal curve. The two extreme end points of the chain of segments form a segment that lies on the convex hull of the input region, called the *closing segment* of the concavity. It follows from the definition of simple regions and the definition of convex hull that the line segments that form the concavity and its closing segment together form a simple region, called the *concavity region*. Our approach is to treat all segments in a concavity as if they have a progress angle equal to their associated closing segment. The effect of such a requirement is that all segments that form a concavity will be processed in successive iterations of Algorithm 1, and each segment in a single concavity will map to a *single* point in the opposing region. The moving segments forming the concavity will not intersect each other, as in Figure 6(b), since all segments in a single concavity will be translated and scaled towards a single point in space and topological relationships between line segments are preserved under translation and scaling. Because this solution is based on Algorithm 1, we maintain the properties that no gaps in the resulting interval region are produced and moving segments do not intersect each other since they are generated based on a convex region. We thus gain the ability to ensure that moving segments forming a single concavity do not intersect each other.

LEMMA 6.1. *The moving segments created from a set of segments at time t_0 that are disjoint or intersect only at end points and that travel to the same point in space p at time t_1 will not intersect except at p or at end points.*

PROOF. The topological relationships between any two line segments hold under the operations of translation, rotation, and scaling. Any group of moving line segments that have the same destination point effectively describe the motion of line segments as they are being translated and scaled to the destination point. If the line segments are disjoint or only intersect at end points at time t, then these topological properties will hold throughout the motion described by the moving line segments. □

Algorithm 2 provides the steps to create an interval region from nonconvex simple regions. Again, the primary change to Algorithm 1 is that progress angles are calculated such that segments in a concavity are assigned the progress angle of their closing segment. To assign the correct progress angles, first the convex hull of a boundary region is computed (Line 1) and an array is created with the convex hull segments in cyclic order beginning with the least most segment (Line 2). A parallel array is created containing the progress angles of the convex hull segments (Line 3). An array of the original region segments ordered in cyclic order beginning with the least most segment is created (Line 4). The array of convex hull segments and original boundary segments are traversed in tandem: if a segment is in both arrays, it is assigned the progress angle from the corresponding progress angle array; if a segment is in the array of segments from the original region and not in the convex hull array that is currently being examined, then it is part of a concavity whose closing segment is the segment in the convex hull array currently being examined, and it is assigned the progress angle of the closing segment (Line 5). The process is repeated for both boundary regions; Algorithm 1 is then executed over the original boundary regions with the progress

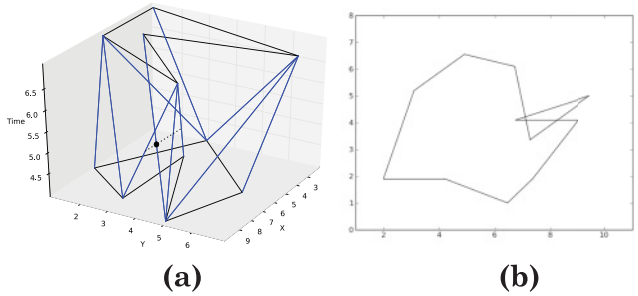


Fig. 7. An example of an invalid interval region generated from valid simple boundary regions (a), and the resulting region at time 5.2 (b). The dot in (a) depicts the initial point of moving segment intersection.

ALGORITHM 2: An Algorithm to Create an Interval Region from Possibly Nonconvex Simple Regions

Input: A source region R and a destination region D
Output: An interval region I represented as a set of moving segments
1 var $Rhull \leftarrow$ the convex hull of R;
2 var $Rhull_{array} \leftarrow$ an array of segments in Rhull in cyclic order beginning with the least most segment in $Rhull$;
3 var $Rhull\phi_{array} \leftarrow$ a parallel array with $Rhull_{array}$ containing the progress angle of the segment sharing an index in $Rhull_{array}$;
4 var $R_{array} \leftarrow$ an array of segments in R in cyclic order beginning with the least most segment in R;
5 var $R\phi_{array} \leftarrow$ a parallel array with R_{array} containing the progress angle of the segment sharing an index in R_{array}. If a segment is not in $Rhull_{array}$, its progress angle is equal to the progress angle of its closing segment in $Rhull_{array}$;
6 Repeat the previous 5 steps for D;
7 Execute Algorithm 1 using the progress angles in $R\phi_{array}$ and $D\phi_{array}$;

angles held in arrays $R\phi_{array}$ and $D\phi_{array}$. Sorting the input arrays in cyclic order takes $O(n \lg n)$ time for n line segments. Convex hull operations can be completed in $O(n)$ time for specific input constraints, and $O(n \lg n)$ in general. Parallel array traversals are linear, and Algorithm 1 simply traverses the input arrays if they are in cyclic order. Thus, Algorithm 2 has time complexity $O(n \lg n)$. $O(n)$ space complexity is achieved because a constant number of arrays are required, and one moving segment will be generated per input segment.

6.2. Interference Between Concavities

Although Algorithm 2 ensures that no two moving segments generated from segments within a single concavity will intersect, it can make no guarantees that moving segments generated from different concavities will not intersect. For example, Figure 7 shows an invalid RIP solution generated by Algorithm 2. Although each individual concavity is handled correctly, different concavities may interact in an invalid manner. In the case of Figure 7, an alternate arrangement of moving segments may be possible to produce a valid interval region; however, the arbitrary complexity of concavities provides the possibility for exotic and complex arrangements. An example of a particularly problematic situation, called the *entwined spiral* configuration, occurs when multiple concavities twist around each other in a spiral pattern (an example is shown in Figure 8). Entwined spirals have the property that moving segments at the end of the spirals, as well as other moving segments forming the spirals, cannot travel

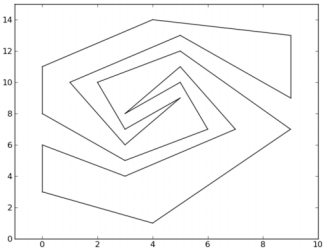


Fig. 8. An example of a region exhibiting an entwined concavity configuration.

across a time interval towards the convex hull of the input region without intersecting another moving segment. Clearly, if an interval region must be created with the simple region in Figure 8 and its convex hull as the boundary regions, the segments at the tips of the spirals are completely surrounded by other segments forming concavities. Furthermore, Algorithm 2 requires all segments in a concavity to travel towards a point on the opposing region's convex hull, guaranteeing moving segment intersections between concavities. Another complication arises from the fact that moving segments form planar delta triangles, meaning that a moving segment cannot change direction in the temporal interior of an interval region. Thus, one cannot simply unwind the spirals over an interval region, especially since spirals can have an arbitrary number of turns. One may attempt to unwind entwined concavities, but each time a moving segment must change direction, a new interval region must be created. Therefore, unwinding a spiral of m turns will require on the order of $O(m)$ interval regions to be created. Each interval region will require its own storage, providing no output size guarantees for the algorithm.

An expressed goal of this article is to produce a general solution to the RIP with defined output size boundaries. Therefore, we take an approach to solving the entwined spirals problem that may not produce the most natural results in some cases, but that is guaranteed to work for arbitrary input simple regions and have a storage bound that is linear with respect to input size.

Note that once an interval region is created by Algorithm 2, one can detect intersecting concavities by finding two delta triangles that intersect in their interiors; such configurations are computable with a triangle/triangle intersection test. Once two delta triangles are known to intersect within an interval region, a general mechanism is required to alter the moving segments such that the intersection is resolved. Recall that untwisting entwined concavities is not desirable since an arbitrary number of untwisting steps may be introduced. Instead, we make the observation that the segments forming a concavity in a simple region along with their closing segment form a simple region that encloses space belonging only to the exterior of the original simple region. Because this space is in the exterior of the original region, any moving segment that traverses this space will not intersect the original boundary segments of the simple region. Therefore, our solution to handling concavities containing moving segments whose delta triangles intersect the delta triangles for some other concavity is to add a new interval region that removes any offending concavities. For example, if the region in Figure 8 is used as a source region and its convex hull used as a destination region to create an interval region, one of the concavities will be removed over an interval region, and a second interval region will be constructed from the end of the first interval region, to the destination region.

To remove a concavity of arbitrary complexity over a single-interval region without introducing new intersecting delta triangles, we allow concavities to, in a sense, *evaporate* in place. Once an offending delta triangle is discovered, all segments from the

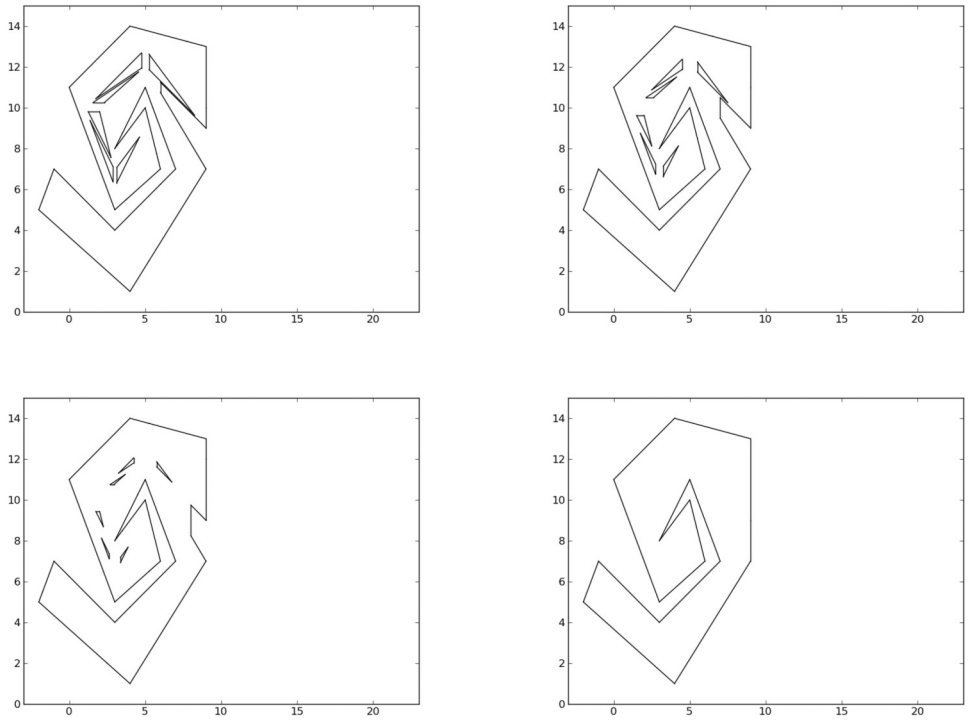


Fig. 9. A concavity evaporation sequence for a snail shell configuration similar to Figure 8.

original region that form the concavity from which that delta triangle was generated, along with their closing segment, are used to create a *concavity region*. That concavity region is then triangulated. For each triangle created, a moving segment is created that travels from each triangle edge to a point within the interior of the triangle. All other segments from the original region simply stay in place over the time interval. Because only segments belonging to offending concavities move over this new interval, and they only move within their respective concavity region, no delta triangle intersections will be introduced; this follows from the fact that in simple regions, all concavity regions will have disjoint interiors. We call the interval region used to remove problematic concavities from a source region the *evaporation step* since concavities visually appear to evaporate. Offending concavities in the destination region use the same process, but reversed, resulting in an *condensation step*, in which the original concavities appear to condense into existence. Figure 9 depicts snapshots of an evaporation step.

Note that multiple concavities may be evaporated/condensed in a single step due to the fact that concavity regions from a single simple region have disjoint interiors. It follows that, at most, a single evaporation step and a single condensation step will be required in order to create valid interval regions depicting the movement of two arbitrary simple regions across a time interval. Recall that a single moving segment is generated for each input segment in Algorithm 1; therefore, the output size for this process is at most $3n$ for n input boundary line segments.

Algorithm 3 details the complete algorithm that produces a valid interval region from a pair of temporally encoded simple regions. Line 1 first uses Algorithm 2 to generate an interval region that possibly contains intersecting delta triangles. In Lines 2 and 3, the concavity regions of the source and destination regions are stored. Practically, these can be computed in conjunction with the convex hull computations of Algorithm 2, thus

they do not add to the time complexity of the algorithm. Lines 6 through 14 test all pairs of delta triangles for intersection. If a pair of delta triangles is found that intersect at any point other than a shared edge or end point, then one of the concavity regions containing one of those triangles is recorded for evaporation or condensation. Lines 17 and 18 compute times between t_1 and t_2 to be used as interval region boundaries for the evaporation and condensation steps, if needed. Lines 28 through 34 create the evaporation interval region. Any concavity regions that must be evaporated are triangulated, and moving segments are created from triangle edges to points within the respective triangles (Lines 28 through 30). Segments from the source region not involved in any triangles are identified (Line 30), and moving segments are created that cause those segments to remain stationary over an interval (Line 31). Finally, all the generated moving segments for the evaporation step are combined into a single-interval region (Line 32). Lines 35 through 42 repeat the procedure for a condensation step. The interval regions are then returned at Line 43.

Lines 1 through 5 of Algorithm 3 use Algorithm 2 and initialize some variables, resulting in an $O(n \lg n)$ time complexity for regions with n segments. Lines 6 through 14 effectively must test all pairs of delta triangles for intersection taking $O(n^2)$. Various methods exists to reduce the running time in practice, but the worst-case complexity remains. Lines 28 and 36 require a triangulation algorithm; general solutions exist that are $O(n \lg n)$. The remainder of the evaporation and condensation generation steps involve set operations, taking at worst $O(n \lg n)$. Therefore, Algorithm 3 has time complexity $O(n^2)$, but in practice is reduced to $O(n \lg n)$ by using data structures to reduce the number of delta-triangle pairs that must be tested for intersection. In terms of space complexity, Algorithm 3 is $O(n)$. The number of interval regions created between arbitrary input regions is limited to 3. Furthermore, segments either do no move over an interval, move to a single point over an interval, or are involved in a single triangle during an evaporation or condensation step. Thus, the number of line segments in each interval is bounded by a constant factor by the number of input segments.

LEMMA 6.2. *Algorithm 3 produces valid interval regions for arbitrary valid input.*

PROOF. Recall that, for an interval region to be invalid, two of its delta triangles must intersect at a point other than an end point or on a shared edge. Algorithm 2 ensures that delta triangles associated with segments in a single concavity will not intersect, and twisting will not occur (Lemma 6.1 and Algorithm 1). Thus, the only violating delta-triangle intersections that can occur belong to delta triangles associated with segments from different concavities, or possibly between a delta triangle associated with a concavity and one associated with a convex hull segment. In either case, Algorithm 3 will remove one concavity associated with a delta triangle from an offending pair. The removal cannot introduce additional delta-triangle intersections since additional moving segments are created that move only in the exterior of the original region, and only within triangles whose interiors do not intersect any other triangles or the interior of the original region. □

7. COMPLEX REGIONS

Complex regions have the ability to represent a single region containing multiple faces and holes; furthermore, complex regions have the ability to exhibit complex structures, such as faces, nested within holes. The existence of multiple, disconnected structures compounds the problems caused by concavity structures in single-region faces introduced in Section 6 by allowing concavities from multiple faces to become intertwined. In this section, we introduce an algorithm that shares certain desirable properties with Algorithm 3: namely, the algorithm will produce moving regions with predictable

ALGORITHM 3: An Algorithm to Create an Interval Region from Arbitrary Simple regions

Input: A source region R at time t_1 and a destination region D at time t_2
Output: A set of interval regions S containing between one and three interval regions

$M \leftarrow$ the moving segments generated by Algorithm 2 using R, t_1, D, t_2 as input;
$R_{concavity\_regions} \leftarrow$ set of concavity regions in R;
$D_{concavity\_regions} \leftarrow$ set of concavity regions in D;
var $evapConcavities \leftarrow \emptyset$;
var $condenseConcavities \leftarrow \emptyset$;
foreach $(r_1, r_2 | r_1, r_2 \in R_{concavity\_regions} \cup D_{concavity\_regions})$ **do**
  if *the delta triangles corresponding to any pair of moving segments, respectively from* r_1 *and* r_2*, intersect* **then**
    if $r_1 \in R_{concavity\_regions}$ **then**
      $evapConcavities \leftarrow evapConcavities \cup r_1$;
    else
      $condenseConcavities \leftarrow condenseConcavities \cup r_1$;
    end
  end
end
if $evapConcavities = condenseConcavities = \emptyset$ **then**
  return M;
var $t_{1.1} | t_1 < t_{1.1} < t_2$;
var $t_{1.2} | t_{1.1} < t_{1.2} < t_2$;
$R_1 \leftarrow R$;
$D_1 \leftarrow D$;
if $evapConcavities = \emptyset$ **then**
  $t_{1.1} \leftarrow t_1$;
end
if $condenseConcavities = \emptyset$ **then**
  $t_{1.2} \leftarrow t_2$;
end
if $evapConcavities \neq \emptyset$ **then**
  $Tris \leftarrow$ triangulate all regions in $evapConcavities$;
  $evap \leftarrow \{m = ((p1_x, p1_y, t_1), (p2_x, p2_y, t_1), (p3_x, p3_y, t_{1.1})) | m \in T \in Tris \ \wedge \ p3 \leftarrow$ an interior point of T $\}$;
  $nonMovingSegs = R - \{s \in C \in evapConcavities\}$;
  $stillSegs =$
  $\{((p1_x, p1_y, t_1), (p2_x, p2_y, t_1), (p1_x, p1_y, t_{1.1})), ((p1_x, p1_y, t_{1.1}), (p2_x, p2_y, t_{1.1}), (p2_x, p2_y, t_1))$
  $|(p1, p2) \in nonMovingSegs\}$;
  $evap \leftarrow evap \cup stillSegs$;
  $R_1 \leftarrow nonMovingSegs \cup \{evapConcavities - R\}$;
end
if $condenseConcavities \neq \emptyset$ **then**
  $Tris \leftarrow$ triangulate all regions in $condenseConcavities$;
  $condense \leftarrow \{m = ((p1_x, p1_y, t_2), (p2_x, p2_y, t_2), (p3_x, p3_y, t_{1.2})) | m \in T \in Tris \ \wedge \ p3 \leftarrow$ an interior point of T $\}$;
  $nonMovingSegs = S - \{s \in C \in condenseConcavities\}$;
  $stillSegs =$
  $\{((p1_x, p1_y, t_{1.2}), (p2_x, p2_y, t_{1.2}), (p1_x, p1_y, t_2)), ((p1_x, p1_y, t_2), (p2_x, p2_y, t_2), (p2_x, p2_y, t_{1.2}))$
  $|(p1, p2) \in nonMovingSegs\}$;
  $condense \leftarrow condense \cup stillSegs$;
  $S_1 \leftarrow nonMovingSegs \cup \{condenseConcavities - S\}$;
end
return $evap$, $condense$, and the result of Algorithm 2 using $R_1, t_{1.1}, S_1, t_{1.2}$;

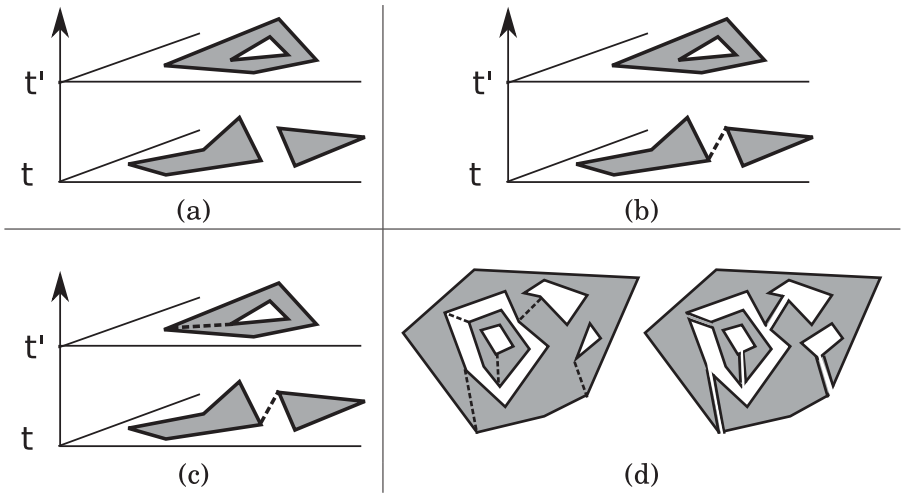


Fig. 10. Two complex regions at times t and t' (a). Dashed external connector segments are shown in (b). Dashed internal and external connector segments are shown in (c). A scene with connector segments and an exaggerated view of their interpretation (d).

bounds on the size of the moving region representation, and the algorithm will work for arbitrary, valid input.

We base our new algorithm that handles complex regions on Algorithm 3. Our approach to building an algorithm is to reduce the RIP for complex regions to an RIP for nonconvex, simple regions, allowing us to use Algorithm 3, which we know to be correct. However, the existence of multiple faces introduces new decisions that must be made by an algorithm. For example, Figure 10(a) depicts two complex regions that may be used as input to an RIP algorithm. This particular example directly raises two questions. First: how should the faces merge into the single face? Second: from where should the hole in the destination region emerge since no hole exists in the source region? In general, there is no single correct answer to these questions without specializing into a specific application domain. Thus, our goal is to provide a generic solution that guarantees correctness; to this end, we provide an $O(n \lg n)$ time reduction of the RIP for complex regions to the RIP for nonconvex, simple regions. The strategy of adapting the problem to fit the input of Algorithm 3 allows us to take advantage of the properties of the algorithm despite the possible complexities imposed by complex regions.

7.1. Defining Connector Segments

In order to fit the problem of complex regions to the input required by Algorithm 3, we first observe that an input containing multiple faces will fail, since the algorithm expects a region defined by a single boundary cycle; therefore, we must *connect* disjoint faces together. To connect faces together, we introduce special segments called *connector segments*. A single connector segment is an imaginary segment that connects two faces. The connector segment is considered to be two parallel segments that are separated by an infinitesimally small distance, creating a single connected face out of two disjoint faces. Figure 10(b) shows the scene from Figure 10(a) with a connector segment connecting the disjoint faces of the source region. There are three restrictions placed on the creation of connector segments:

(1) Connector segments cannot intersect other segments, or other connector segments, in their interiors.
(2) Connector segments must share end points with segments from two disjoint cycles in a region.
(3) Connector segments must be introduced such that the boundaries of previously disconnected cycles become a single cycle such that a traversal of the boundary in cyclic order visits every boundary segment (as required by Algorithm 3). In other words, if cycles are considered vertices and connector segments are considers edges in graph G, G must be acyclic.

The insertion of connector segments between region faces effectively treats the space between a pair of disjoint faces as a concavity, which is easily handled by Algorithm 3. Holes will be handled in a similar fashion. According to the definition of complex regions, a hole will occur in exactly one face; thus, we introduce a connector segment in the interior of a face that connects the outer boundary of that face to the hole. These connector segments lie in the interior of a region, thus they represent two imaginary, parallel lines that are separated by an infinitesimally small distance, and the region's exterior lies between these imaginary lines. Effectively, connector segments that lie on a region's interior cause the connected hole to be treated as a concavity. Connector segments can connect a face to an enclosing hole or two holes that lie in the same face. Figure 10(c) shows two complex regions with valid connector segment placement. Figure 10(d) shows a more structurally complex scene with a valid placement of connector segments and an exaggerated view of the interpretation of the region's boundary implied by the connector segments. Note the existence of a single cyclic order for traversing the entire structure, as required by Algorithm 3, and the treatment of holes and contained faces as concavities. We define connector segments in general as follows:

Definition 7.1. Connector segments are imaginary segments introduced into a complex region to connect disjoint cycles such that the complex region can be used as input to Algorithm 3. Correct placement of connector segments on a region results in a region R and a set of connector segments C with the following properties:

(1) A connector segment cannot intersect any other segment or connector segment in its interior.
(2) Let $G = (V, E)$ be the graph, where V is the set of all segment end points in R and E is the set of edges formed by segments in the R. Each end point of a connector segment $c \in C$ must lie on end point s_p of segment $s \in R$ and end point u_p of segment $u \in R$, respectively, such that s_p and u_p are not connected in G.
(3) Let $F = (V, E)$ be the graph in which each edge $v \in V$ represents a connected component in G, as defined earlier, and each $e \in E$ corresponds to a connector segment in C that connects two vertices in V. F is acyclic and connected.

Note that, according to Definition 7.1, connector segments are not needed between two distinct cycles that share an end point; essentially a zero length connector segment exists connecting the two end points, thus no additional segments are required.

7.2. Computing Connector Segments

For a region R, many correct placements of connector segments are possible. We provide an $O(n \lg n)$ algorithm based on the plane sweep technique that creates a valid configuration of connector segments for an input region. Note that a plane sweep algorithm requires an input consisting of *halfsegments*: for each segment $s \in R$, two halfsegments exist $h = (s_p, s_q)$ and $g = (s_q, s_p)$. The first point in the tuple defining a halfsegment

is called the *dominating point* and the other point is called the *submissive point*. For halfsegment h, we use the notations h_d and h_s, to respectively denote its dominating and submissive points. A halfsegment h is a *left halfsegment* if $h_d < h_s$; otherwise, it is a *right halfsegment*. The left and right halfsegment corresponding to the same segment are called *brothers*. Let $\theta(h)$ be the angle formed by rotating h around h_d in the counterclockwise direction until it is vertical and extending upwards in the y direction from h_d. A total ordering exists over halfsegments:

$$h < g \Leftrightarrow \begin{pmatrix} h_d < g_d \vee \\ (h_d = g_d \wedge \theta_h < \theta_g) \vee \\ (h_d = g_d \wedge \theta_h = \theta_g \wedge h_s < g_s) \end{pmatrix} \tag{4}$$

A region R can be represented as a list of segments or as a list of halfsegments. The halfsegment representation provides a straightforward mechanism to identify unique cycles in a region. Due to halfsegment ordering, when traversing an ordered halfsegment list representing a region, the first halfsegment h encountered for any cycle C will be a left halfsegment, and the interior of the simple region induced by C will lie above that halfsegment. Therefore, to find the next halfsegment in cyclic order, we must rotate h around its submissive point through the interior of C until a new halfsegment j is encountered; j represents the next segment in cyclic order around C. Given h, j can be found in $O(\lg n)$ time for a region containing n segments by constructing h's brother halfsegment g, doing a binary search for g on the halfsegment list. j will share a dominating point with g, and can be found by examining g's neighbors in the halfsegment list. Using this process, we identify all valid cycles in R using Algorithm 4. Algorithm 4 is a simplified version of the algorithm described in McKenney [2009].

ALGORITHM 4: An Algorithm to Identify All Unique Cycles in a Region

Input: A source region R defined as an array of ordered halfsegments, and a parallel array of integers L.
Output: The i_{th} location of L will contain a cycle identifier indicating to which cycle the i_{th} halfsegment of R belongs.

```
1  Initialize L to contain zeros;
2  var id = 1;
3  foreach i ∈ {0, ..., length(R)} do
4      if L[id] = 0 then
           // Found an unprocessed halfsegment
           // This is the first halfsegment of some cycle
5          var firstIndex ← i;
6          repeat
7              var brother ← (R[i]_s, R[i]_d);
8              var brotherIndex ← find index of brother in R;
9              L[i] = id;
10             L[brotherIndex] = id;
11             i ← index of the next halfsegment found when rotating brother clockwise
                   around brother_d;
12         until i ≠ firstIndex;
13         id ← id + 1;
14     end
15 end
```

Note that Algorithm 4 treats every cycle as if it is an outer cycle (as opposed to a hole cycle); a step is required to determine which cycles are holes and which are outer

cycles. A basic plane sweep algorithm [Shamos and Hoey 1976; Bentley and Ottmann 1979] will make this determination.

Algorithm 4 assigns a unique integer *identifier* to each cycle. The next step is to assign cycles that share boundary points a single, unique identifier; we call multiple cycles that are connected through shared boundary points a *connected structure* in the region. This process is straightforword: we simply scan the halfsegment array to find instances in which multiple halfsegments share an end point but have different cycle identifiers. We use a hash table to map the original cycle numbers of cycles in a connected structure to a new, unique cycle number. We then scan the halfsegment array and cycle identifier array to assign halfsegments to the correct cycle identifier using the hash table.

Finally, connector segments must be placed to connect disconnected cycles in the input region. Again, we will use a plane sweep algorithm. Recall that the sweep line algorithm sweeps an imaginary line, called the *sweep line*, across the plane; the line can sweep in any direction, as long as the line is perpendicular to the sweeping direction. We assume that the sweep line is vertical (in the y direction) and sweeps horizontally (in the x direction) across the plane. The plane sweep algorithm maintains an *active list* that stores the left halfsegments that currently intersect the sweep line sorted in the order at which they intersect the sweep line. Our algorithm will maintain a Boolean array indexed by the cycle numbers used in an input region, and with all values initialized to *false*. When a connected structure in an input region is connected to another connected structure, the array locations specified by the connected structures' cycle numbers will be set to *true*; we use this array to enforce the third property of valid connector segment placement presented earlier. A connector segment can be added only between pairs of connected structures such that one structure has a value of *false* in the array.

Algorithm 5 describes the addition of connector segments to a region with cycles identified. The algorithm to compute connector segments must handle three cases. The first case occurs when two connected structures are linearly separable by a vertical line (Lines 10 through 13). In this case, the greatest segment end point of the leftmost connected structure and the least most end point of the rightmost connected structure form the end points of a connector segment. Such a case is identified when the number of halfsegments in the active list drops to zero. Due to halfsegment ordering, a connector segment constructed from the indicated end points will not intersect any other segment or segment endpoint. Note that due to halfsegment ordering, this procedure correctly inserts vertical connector segments for connected structures that are not linearly separable in the y direction, but have a greatest boundary point in one connected structure with the same y value as the least boundary point in another connected structure. Because this situation represents the first time the rightmost structure is encountered, a connector segment is always added and the Boolean array is updated to indicate that both connected structures involved are now connected.

The second case that the algorithm must handle is when the first halfsegment h of a connected structure with identifier n is added to the active list, and a halfsegment j belonging to another connected structure with identifier m is identified directly below the dominating point h_d (Lines 14 through 19). Due to halfsegment ordering and the definition of complex regions, no other halfsegments exist that are not already in the active list that are below h_d and that cross the vertical line extending through h_d. Therefore, we can safely add a connector segment from h_d to the point at which a vertical line through h_d intersects j. Again, a connector segment is added only if one of the involved connected structures has not yet been connected, and the Boolean array is updated accordingly.

The third case that the algorithm must handle is when the first halfsegment h of a connected structure with identifier n is added to the active list, no halfsegments lie

below h in the active list, and a halfsegment j from another connected structure with identifier m is found in the active list directly above dominating point h_d (Lines 20 through 36). Let l be a vertical line extending through h_d. It follows from the definition of complex regions and halfsegment ordering that additional halfsegments with a dominating point on l may exist that are not yet in the active list; therefore, we must look ahead in the halfsegment array to find j', the smallest halfsegment belonging to a connected cycle with (i) identifier $m' \neq n$, (ii) a dominating point lying on l, and (iii) a dominating point that intersects l below the point at which j intersects l. If no such halfsegment is found, then $j' \leftarrow j$. Similarly, we must find the largest halfsegment h' (i) from the connected structure with identifier n, (ii) with a dominating point on l, and (iii) with a dominating point that lies below point where j' intersects l. If no such halfsegment is found, then $h' \leftarrow h$. Both h' and j' can be located in the same scan. If one of the connected structures that includes h' or j' have a value of *false* in the Boolean array (i.e., one structure is not yet connected), then a connector segment is created using h'_d and the intersection point of j' and l as its end points, and the Boolean array is updated.

LEMMA 7.2. *Algorithm 5 correctly creates connectors corresponding to Definition 7.1.*

PROOF. Due to half segment ordering and the definition of complex regions, the first left halfsegment h belonging to a cycle C that is encountered in halfsegment order will be a left halfsegment, and its dominating point will be the least most point on that C's boundary; furthermore, no connector segments involving C will yet be generated. Thus, if h is the only segment in the active list and halfsegments have been removed from the active list, a connector segment between the dominating point of h and the submissive point of the last left halfsegment removed from the active list cannot intersect any other segments, and will connect C to the linearly separable cycle that precedes it in the x direction (Case 1). If a halfsegment g lies below h in the active list, then due to halfsegment ordering, no other segments can exist between g and h, thus a vertical connector segment between them cannot intersect any other segment or connector segment (Case 2). If halfsegment g lies above h in the active list, it is possible that any number of segments may exist that are not yet in the active list with a dominating point on a vertical line between g and h (Case 3). In this case, we must look ahead in the region list to find the greatest $h' \geq h$ with identical x value in the dominating point as h that belongs to the same cycle as h in an unbroken sequence of segments from the same cycle as h. We must then check if the next greater halfsegment is g, or a segment g' that shares a dominating point x value with h and is from a different cycle than h. If g' exists, a connector segment is created between its dominating point and h's dominating point; otherwise, a vertical connector segment is created from h's dominating point to g. According to halfsegment ordering, the new connector cycle cannot intersect any other segments in the region. All cases that may exist when adding a halfsegment to the active list are then exhausted. Following this procedure ensures that connector segments are constructed between vertically adjacent segments, therefore they cannot intersect with each other, or a single connector segment is made between linearly separable regions in the x direction. Because connectors between linearly separable regions always involve the greatest end point of the most recently removed cycle, no two generated connector segments between linearly separable regions in the x direction can intersect. It follows that all connector segments conform to Definition 7.1. □

Algorithm 5 is a modification of a plane sweep algorithm that does not find line segment intersections; therefore it is based on an $O(n \lg n)$ structure for a region with n line segments. The *while* loop on Line 21 is the only modification that could increase the time complexity. Because the *if* statement on Line 7 is entered only for halfsegments

ALGORITHM 5: Algorithm to Place Connector Segments on a Region

Input: A region R defined as an array of ordered halfsegments, a parallel array of integers L indicating the identifier of the connected structure to which each halfsegment belongs, and and a Boolean array B that is indexed by the connected structure identifiers used in R and stored in L.
Output: A list C of connector segments for R.

Initialize L using Algorithm 4 and the procedure using a hash table to identify connected structures;
var *lastOutOfALIndex* $\leftarrow$ *NULL*;
var *conn*;
foreach $i \in \{0, \ldots, \text{length}(R)\}$ **do**
  $h \leftarrow R[i]$;
  if *h is left and* $B[L[i]]$ **then**
    add h to the active list;
  else if *h is left and* $\neg B[L[i]]$ **then**
    `// Case 1`
    add h to the active list;
    if *active list contains only h and* lastOutOfALIndex $\neq$ *NULL* **then**
      $conn \leftarrow (R[lastOutOfALIndex]_s, h_d)$;
      append *conn* to C;
      $B[L[i]] \leftarrow B[L[lastOutOfALIndex]] \leftarrow true$;
    `// Case 2`
    else if *halfsegment* $j = R[x]$ *is below h in the active list* **then**
      $l \leftarrow$ vertical line through h_d;
      $p \leftarrow$ intersection of l and j;
      $conn \leftarrow (p, h_d)$;
      append *conn* to C;
      $B[L[i]] \leftarrow B[L[j]] \leftarrow$;
    `// Case 3`
    else if *halfsegment* $j = R[x]$ *is below h in the active list* **then**
      `// If an unbroken series of halfsegments exists that are from the same connected structure as` h `and whose dominating point lies on the sweep line, find the largest halfsegment in the series.`
      $h' \leftarrow h$;
      $y = i + 1$;
      while $h'_{dx} = R[y]_{dx} \wedge L[i] = L[y]$ **do**
        $h' \leftarrow R[y]$;
        $y \leftarrow y + 1$;
      `// If a halfsegment exists that is from a different connected structure as` h' `and whose dominating point lies on the sweep line and is next in halfsegment order from` h'`, then connect to that halfsegment.`
      $j' \leftarrow j$;
      $z \leftarrow x$;
      if $h'_{dx} = R[y + 1]_{dx}$ **then**
        $j' = R[y + 1] z = y + 1$
      $l \leftarrow$ vertical line through h'_d;
      $p \leftarrow$ intersection of l and j';
      $conn \leftarrow (h'_d, p)$;
      append *conn* to C;
      $B[L[y]] \leftarrow B[L[z]] \leftarrow true$;
  else if *h is right* **then**
    remove the brother of h from the active list;
    lastOutOfALIndex $\leftarrow i$;

belonging to a connected structure that has not been used to create connected segments, the *while* loop on Line 21 will be accessed at most once per connected component, and can visit at most the number of halfsegments in that connected component. The number of connector segments generated is bounded by the number of cycles in a region, which is bounded by the number of input segments; thus, the $O(n \lg n)$ time complexity for a region with n segments is preserved. The plane sweep algorithm is also used to identify unique cycles, resulting in an $O(n \lg n)$ procedure to convert complex region input to a format compatible with Algorithm 3.

Finally, the complete solution to the RIP for arbitrary complex regions S at time t_1 and D at time t_2 is to identify unique cycles, run Algorithm 5 to insert connector segments, and then run Algorithm 3. Algorithm 3 must have minor modifications in order for it to interpret the cyclic order of input region boundaries correctly based on connector segments.

THEOREM 7.3. *The proposed solution to the RIP for arbitrary complex regions returns at most 3 valid interval regions describing the motion between a source region and destination region from time t_1 to time t_2*

PROOF. Lemma 7.2 creates valid connector segments that transform a complex region into a nonconvex, simple region. If the input region is already simple, then no connector segments are created. Lemma 6.2 ensures that valid interval regions are created for nonconvex or convex simple input regions. □

Figure 11 depicts a sequence of images showing the result of the proposed solution to the RIP, generating three interval regions to describe the movement between two snapshots. The snapshots contain holes, faces within holes, and complex concavity structures. The delta triangles generated by the algorithm are plotted in three dimensions in Figure 12.

8. EXPERIMENTS

In order to verify the theoretical work proposed in this article, we have implemented the RIP algorithm and used that implementation to generate many of the figures used here. We implemented the algorithm in the Python language. The intent of this particular implementation is to provide a portable, readable reference implementation of the algorithm that others can use or transcribe into other languages; however, the implementation does provide insights into running-time behavior for various input. The implementation has been integrated into the *pyspatiotemporalgeom* package hosted at the Python Package Index [McKenney 2014a], and the source code is available in McKenney [2014b]. We ran all experiments on a desktop computer with 4GB RAM and an Intel Core i7 processor running at 3.4GHz.

Figure 13 depicts the running time of the RIP algorithm for five pairs of input regions of increasing size. The input regions used are randomly generated, and contain rather complex structures that exhibit multiple concavities, faces, and holes. For example, the smallest pair of input regions (in terms of number of boundary-line segments) is shown in Figure 14. The quadratic nature of the curve is visible; however, running time of the algorithm is dominated by a small portion of the algorithm: the computation of triangle/triangle intersections in Algorithm 3, Lines 6 and 7, in which the delta triangles of concavities are tested for intersection. In our implementation, this portion of the algorithm is written with time complexity $O(n^2)$, where n is the number of delta triangles computed between two regions. Advanced techniques and data structures exist to reduce the theoretical time complexity to $O(n \lg n)$ [Gottschalk et al. 1996], and many optimizations are available for geometric intersections, but they were avoided in order to provide a concise, portable, and readable reference implementation. For the

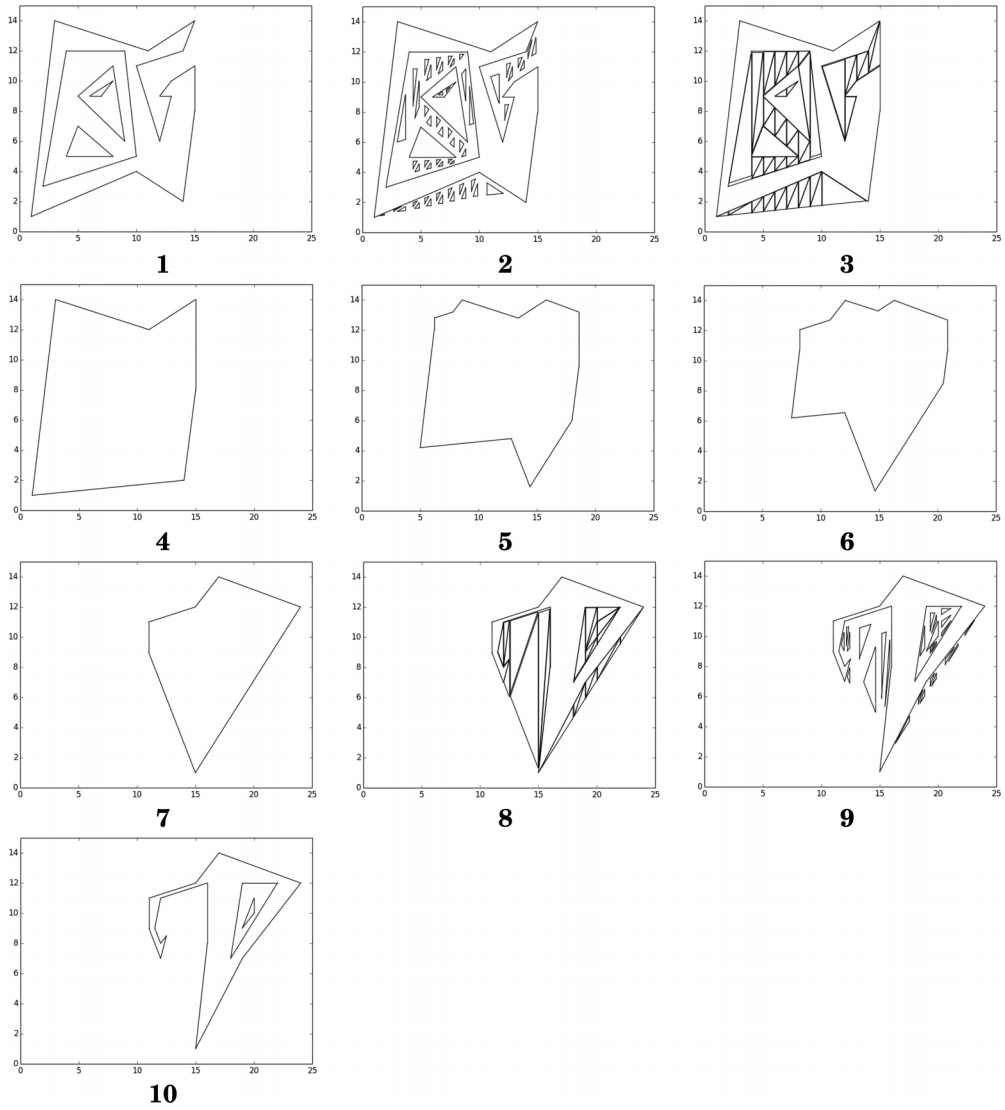


Fig. 11. The result of the proposed algorithm creating a valid moving region between the snapshots at either end of the sequence. This particular result evaporates troublesome concavities in an interval region, creates an intermediate interval region, and condenses troubling concavities in a final interval region. The three-dimensional view of the moving region is shown in Figure 12. Numbers indicate the progression of snapshots.

smallest experiment, triangle/triangle intersection tests consume 87.5% of execution time, growing to 98.9% in the largest experiment. If the time consumed by triangle/triangle intersections is removed, the running times display the expected $O(n \lg n)$ curve, as shown in Figure 15.

The purpose of the triangle/triangle intersection routine in the RIP algorithm is simply to test for moving segments that may intersect within an interval, causing a moving region to have a self-intersecting boundary at some point in time. If no delta triangles intersect, then the algorithm terminates; if intersecting delta triangles are found, the

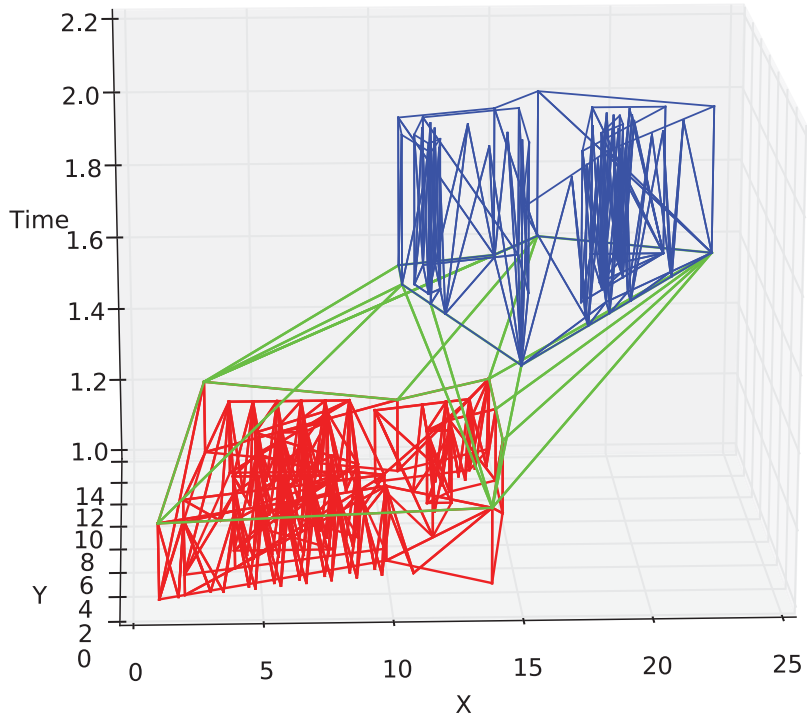
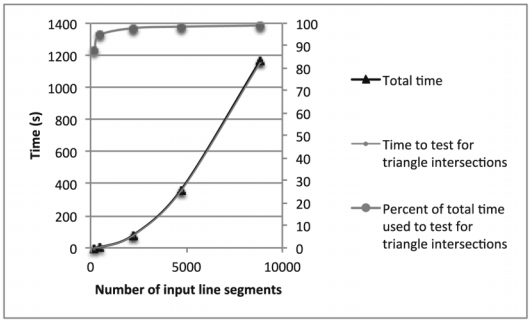


Fig. 12. The result of the proposed algorithm creating a moving region between the snapshots at both ends of the sequence. A sampled progression of this moving region is depicted in Figure 11 Three interval regions are used to construct the final moving region. The three-dimensional view of the moving region is depicted.



Number of input segments	Total time (s)	Triangle/triangle intersection time (s)	Percent of time computing triangle/triangle intersections
217	0.8	0.7	87.5%
480	3.7	3.5	94.6%
2220	78.6	76.8	97.7%
4730	362.8	356.9	98.4%
8836	1165.8	1152.8	98.9%

Fig. 13. Running time of the RIP algorithm for inputs or various sizes. For each run, the total time to interpolate is plotted, along with the time taken to compute triangle/triangle intersections. The upper curve represents the percent of total running time dedicated to computing triangle/triangle intersections. The number of input line segments reported is the sum of the line segments forming the boundaries of both input regions. In each experiment, the source and destination region contained similar numbers of line segments.

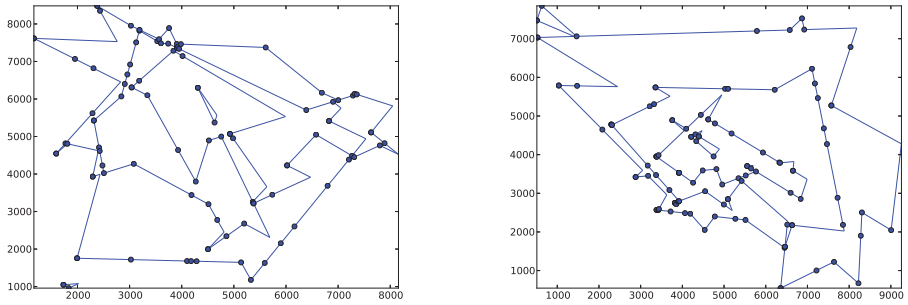


Fig. 14. The pair of regions containing a combined 217 segments used to generate running times for Figure 13. The region on the left contains 101 line segments; the region on the right contains 116 line segments.

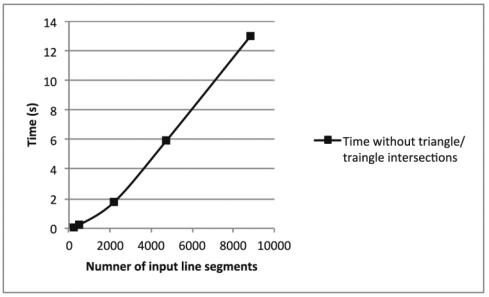


Fig. 15. The running time of the experiments in Figure 13, not including the time to compute triangle/triangle intersections.

evaporation and condensation stages are added, and the original algorithm is simply run again without the problematic concavities. Therefore, the behavior of the algorithm on large datasets can be estimated by simply skipping the triangle/triangle intersection test. In fact, if the input is known to not have any concavities that will cause intersecting delta triangles, the triangle/triangle intersection test is unnecessary. We ran two experiments in which the algorithm terminates just before the triangle/triangle intersection test. The first experiment uses the boundary of the state of California as the earlier interval region boundary and the boundary of the state of Texas as the later boundary. The state of California is represented using 11,648 line segments; the state of Texas is represented using 35,661 line segments. The RIP implementation required 2.9s to create an interval region between the two states.

The second experiment omitting the triangle/triangle intersection test used the HURDAT hurricane dataset [Demuth et al. 2006; NOAA 2014]. The HURDAT dataset provides the extent of hurricane force winds for hurricanes as snapshots at specified time instants. For example, a hurricane may have hurricane-force winds for a radius of 2 miles centered at a particular coordinate at noon, then have a radius of hurricane force winds extending for 4mi at 6:00pm centered at another coordinate, and so on. We use our algorithm to create interval regions between the snapshots. Each hurricane snapshot is represented using a small number of line segments, usually around 20. The dataset contains 2,063 pairs of regions used to create interval regions, with a total of 105,222 line segments among the pairs. Interpolating between all pairs, effectively creating moving regions from the hurricane snapshots, took 2.8s.

The experimental data used in this section included randomly generated regions with a highly complex structure and geographic regions with a more simple structure. The

experiments reflect the fact that the number and complexity of concavities, holes, and multiple faces in an input have little effect on execution time; effectively, each hole and face requires only a single line segment to be added to the computation. Furthermore, the duration of the interval between a pair of snapshots is irrelevant since it only affects the height of delta triangles in the temporal dimension. In fact, the hurricane dataset used snapshots over 6-hour durations and the generated data used snapshots with durations of a few seconds. In either case, the duration between snapshots had no effect on experiments. Finally, the sampling rate of snapshots has no effect on the execution of our algorithm since it is defined on interval regions. A higher sampling rate results in more interval regions, and the algorithm must be run over more pairs of input regions. Because our algorithm produces an approximation of a region during a time interval, a higher sampling rate will cause the resulting moving region to be a closer approximation of the actual moving region.

These experiments show that the algorithm performs well enough to be useful even in the unoptimized state. As is common when converting a Python script to an optimized C program, we expect significant speedups for an optimized implementation.

9. CONCLUSION

In this article, we define the region interpolation problem for moving regions and differentiated it from polygonally morphing problems encountered in computer graphics applications. We provided the first solution to the RIP that creates valid interval regions for arbitrary, valid input consisting of complex regions. The proposed algorithm has a worst-case time complexity of $O(n^2)$, but can be achieved in $O(n \lg n)$ in practice for input regions containing n segments. The algorithm requires $O(n)$ space, and produces an output with size bounded by $O(n)$. Because a single, fixed interpolation scheme is used between snapshots, users must insert additional snapshots to control the movement of regions at a finer granularity; however, the linear bound on output size provides predictable size guarantees that users can utilize in determining an appropriate level of snapshot granularity.

The concept of moving regions is applicable to a wide range of phenomena in the real world. As stated in Section 1, rich data sources exist from which moving-region data can be extracted. For example, a region representing the extent of a hurricane can be extracted from a satellite image at a particular instant. Our RIP algorithm can then automatically create a moving region out of a series of such snapshots. Without the algorithm presented in this article, no automated mechanism of creating a moving region from snapshots that guaranteed correctness of the resulting data type existed; such interpolations would need to be made by hand. Because there are many imaging sources available (e.g., radar, multispectral imaging, lidar), the algorithm presented here creates possibilities to generate datasets that were not previously possible.

REFERENCES

M. Alexa. 2002. Recent advances in mesh morphing. *Computer Graphics Forum* 21, 173–198. DOI:http://dx.doi.org/10.1111/1467-8659.00575

M. Alexa, D. Cohen-Or, and D. Levin. 2000. As-rigid-as-possible shape interpolation. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'00)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, 157–164. DOI:http://dx.doi.org/10.1145/344779.344859

P. Alliez, G. Ucelli, C. Gotsman, and M. Attene. 2008. Recent advances in remeshing of surfaces. In *Shape Analysis and Structuring*, L. De Floriani and M. Spagnuolo (Eds.). Springer, Berlin, 53–82. DOI:http://dx.doi.org/10.1007/978-3-540-33265-7_2

J. Barbic, F. Sin, and E. Grinspun. 2012. Interactive editing of deformable simulations. *ACM Transactions on Graphics* 31, 4, Article 70, 8 pages. DOI:http://dx.doi.org/10.1145/2185520.2185566

J. L. Bentley and T. A. Ottmann. 1979. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers,* C-28, 9, 643–647. DOI:http://dx.doi.org/10.1109/TC.1979.1675432

E. Carmel and D. Cohen-Or. 1998. Warp-guided object-space morphing. *The Visual Computer* 13, 9–10, 465–478. DOI:http://dx.doi.org/10.1007/s003710050118

J. Demuth, M. DeMaria, and J. A. Knaff. 2006. Improvement of advanced microwave sounder unit tropical cyclone intensity and size estimation algorithms. *Jounral of Applied Meteorology* 45, 1573–1581.

M. J. Egenhofer and J. Herring. 1990. *Categorizing Binary Topological Relations Between Regions, Lines, and Points in Geographic Databases*. Technical report. National Center for Geographic Information and Analysis, University of California, Santa Barbara.

A. U. Frank, M. J. Egenhofer, and J. P. Jackson. 1990. A topological data model for spatial databases. In *Symposium on the Design and Implementation of Large Spatial Databases.* Springer-Verlag, New York, NY.

A. Frank and W. Kuhn. 1986. Cell graph: A provable correct method for the storage of geometry. In *Proceedings of the Second International Symposium on Spatial Data Handling*. 411–436.

C. Gotsman and V. Surazhsky. 2001. Guaranteed intersection-free polygon morphing. *Computers and Graphics* 25, 1, 67–75. DOI:http://dx.doi.org/10.1016/S0097-8493(00)00108-4 Shape Blending.

S. Gottschalk, M. C. Lin, and D. Manocha. 1996. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'96)*. 171–180.

R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. 2000. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems* 25, 1, 1–42.

J. Herring (Ed.). 2010. OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option.

A. W. F. Lee, D. Dobkin, W. Sweldens, and P. Schröder. 1999. Multiresolution mesh morphing. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'99)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, 343–350. DOI:http://dx.doi.org/10.1145/311535.311586

J. Lema, L. Forlizzi, R.H. Güting, E. Nardelli, and M. Schneider. 2003. Algorithms for moving objects databases. *Computer Journal* 46, 6.

M. McKenney. 2009. Region extraction and verification for spatial and spatio-temporal databases. In *SSDBM*, *(Lecture Notes in Computer Science*. Springer, 598–607.

M. McKenney. 2014a. PySpatioTemporalGeom Package, Version 0.2. Retrieved June 29, 2015 from https://pypi.python.org/pypi/pyspatiotemporalgeom/.

M. McKenney. 2014b. PySpatioTemporalGeom Source Code. Retrieved June 29, 2015 from https://bitbucket.org/marmcke/pyspatiotemporalgeom/.

M. McKenney and J. Webb. 2010. Extracting moving regions from spatial data. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'10)*. ACM, New York, NY, 438–441. DOI:http://dx.doi.org/10.1145/1869790.1869856

NOAA. 2014. HURDAT Re-analysis Project. (2014). Retrieved June 29, 2015 from http://www.aoml.noaa.gov/hrd/hurdat/Data_Storm.html.

M. Schneider and T. Behr. 2006. Topological relationships between complex spatial objects. *ACM Transactions on Database Systems* 31, 1, 39–81. DOI:http://dx.doi.org/10.1145/1132863.1132865

T. W. Sederberg, P. Gao, G. Wang, and H. Mu. 1993. 2DD shape blending: An intrinsic solution to the vertex path problem. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'93)*. ACM, New York, NY, 15–18. DOI:http://dx.doi.org/10.1145/166117.166118

T. W. Sederberg and E. Greenwood. 1992. A physically based approach to 2-D shape blending. *SIGGRAPH Computer Graphics* 26, 2, 25–34. DOI:http://dx.doi.org/10.1145/142920.134001

M. I. Shamos and D. Hoey. 1976. Geometric intersection problems. In *17th Annual Symposium on Foundations of Computer Science*. 208–215. DOI:http://dx.doi.org/10.1109/SFCS.1976.16

M. Shapira and A. Rappoport. 1995. Shape blending using the star-skeleton representation. *IEEE Computer Graphics and Applications* 15, 2, 44–50. DOI:http://dx.doi.org/10.1109/38.365005

A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. 1997. Modeling and querying moving objects. In *13th International Conference on Data Engineering*, IEEE.

E. Tossebro and R. H. Guting. 2001. Creating representations for continuously moving regions from Observations. In *Advances in Spatial and Temporal Databases*, C. S. Jensen, M. Schneider, B. Seeger, and V. J. Tsotras (Eds.). *Lecture Notes in Computer Science*, Vol. 2121. Springer, Berlin, 321–344. DOI:http://dx.doi.org/10.1007/3-540-47724-1_17

G. Trajcevski, O. Wolfson, K. Hinrichs, and S. Chamberlain. 2004. Managing uncertainty in moving objects databases. *ACM Transactions on Database Systems* 29, 3, 463–507. DOI:http://dx.doi.org/10.1145/1016028.1016030

O. van Kaick, H. Zhang, G. Hamarneh, and D. Cohen-Or. 2011. A survey on shape correspondence. *Computer Graphics Forum* 30, 6, 1681–1707. DOI:http://dx.doi.org/10.1111/j.1467-8659.2011.01884.x

O. Wolfson, P. Sistla, B. Xu, J. Xu, and S. Chamberlain. 1999. DOMINO: Databases for moving objects tracking. *SIGMOD Record* 28, 2, 547–549. DOI:http://dx.doi.org/10.1145/304181.304572

M. F. Worboys. 1994. A unified model for spatial and temporal information. *Computer Journal* 37, 1, 26–34. DOI:http://dx.doi.org/10.1093/comjnl/37.1.26

W. Yang and J. Feng. 2009. 2D shape morphing via automatic feature matching and hierarchical interpolation. *Computers & Graphics* 33, 3, 414–423. DOI:http://dx.doi.org/10.1016/j.cag.2009.03.007

W. Yang, J. Feng, and X. Wang. 2012. Structure preserving manipulation and interpolation for multi-element 2D shapes. *Computer Graphics Forum* 31, 7, 2249–2258. DOI:http://dx.doi.org/10.1111/j.1467-8659.2012.03218.x

Received March 2014; revised September 2014; accepted December 2014