Dynamic Ray Shooting and Shortest Paths via Balanced Geodesic Triangulations

(Preliminary version)

Michael T. Goodrich*
Dept. of Computer Science
The Johns Hopkins University
Baltimore, MD 21218
goodrich@cs.jhu.edu

Roberto Tamassia†
Dept. of Computer Science
Brown University
Providence, RI 02912–1910
rt@cs.brown.edu

Summary of Results

We give new methods for maintaining a data structure that supports ray shooting and shortest path queries in a dynamically-changing connected subdivision $\mathcal{S}$. Our approach is based on a new dynamic method for maintaining a balanced decomposition of a simple polygon via geodesic triangles. We maintain such triangulations by viewing their dual trees as balanced trees. We show that rotations in these trees can be implemented via a simple "diagonal swapping" operation performed on the corresponding geodesic triangles, and that edge insertion and deletion can be implemented on these trees using operations akin to the standard *split* and *splice* operations. We also maintain a dynamic point location structure on the geodesic triangulation, so that we may implement ray shooting queries by first locating the ray's endpoint and then walking along the ray from geodesic triangle to geodesic triangle until we hit the boundary of some region of $\mathcal{S}$. The shortest path between two points in the same region is obtained by locating the two points and then walking from geodesic triangle to geodesic triangle either following a boundary or taking a shortcut through a common tangent. Our data structure uses $O(n)$ space and supports queries and updates in $O(\log^2 n)$ time, where n is the current size of $\mathcal{S}$. It outperforms the previous best data structure for this problem by a $\log n$ factor in all the complexity measures (space, query times, and update times).

*This research was supported in part by the National Science Foundation under Grant CCR-9003299, and by the NSF and DARPA under Grant CCR-8908092.

9th Annual Comptutational Geometry,5/93/CA,USA
© 1993 ACM 0-89791-583-6/93/0005/0318...$1.50

1 Introduction

An exciting trend in algorithmic research has been to show how one can efficiently maintain various properties of a combinatoric or geometric structure while updating that structure in a dynamic fashion; see, for example [5, 8, 9, 11, 19, 21, 23, 26].

1.1 The Problem

The specific dynamic computational geometry problem we address in this paper is to maintain a connected subdivision $\mathcal{S}$ subject to insertion and deletion of vertices and edges, and to ray shooting and shortest path queries. From now on, we denote with n the current size of $\mathcal{S}$, i.e., the number of vertices of $\mathcal{S}$.

1.2 Previous Work

In the static setting, there are several optimal techniques for shortest-path and ray-shooting [1, 3, 6, 12, 13, 18], even in parallel [10, 15]. In particular, the data structures of Chazelle and Guibas [3] and of Guibas and Hershberger [12] support respectively ray-shooting and shortest path queries in simple polygons in $O(\log n)$ time using $O(n)$ space. In the dynamic setting, the best result to date is the data structure of Chiang, Preparata, and Tamassia [4] for connected subdivisions, which uses $O(n \log n)$

†This research was supported in part by the National Science Foundation under grant CCR-9007851, by the U.S. Army Research Office under grant DAAL03-91-G-0035, and by by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052, ARPA order 8225.

space and supports ray-shooting queries, shortest path queries, and insertion and deletion of vertices and edges in $O(\log^3 n)$ time (amortized for vertex updates).

1.3 Our Results

In this paper we present a dynamic data structure for connected subdivisions that supports ray-shooting and shortest-path queries. The repertory of of update operations includes insertion and deletion of vertices and edges, and is complete for connected subdivisions. The space requirement is $O(n)$, and the worst-case running time for all operations (queries and updates) is $O(\log^2 n)$, where n is the current size of the subdivision. Our data structure outperforms the previous best data structure for this problem by a $\log n$ factor in all the complexity measures (space, query times, and update times). Also, it is conceptually simple.

1.4 Overview of the Technique

A *geodesic* path between two points p and q inside a simple polygon P is the shortest path joining p and q that does not go outside P. Given three vertices u, v, and w of a simple polygon P, which occur in that order, the *geodesic triangle* $\tilde{\triangle}uvw$ they determine is the union of the geodesic paths from u to v, from v to w, and from w to u. (See Figure 1.) A *geodesic triangulation* of a simple polygon P is a decomposition of P's interior into geodesic triangles whose boundaries do not cross. Two geodesic triangles may have a non-empty intersection, however, if portions of their respective boundaries overlap.

A geodesic triangulation is combinatorially and topologically like a triangulation of a simple polygon. Hence, it immediately induces a degree-3 tree T, where each node in T corresponds to a geodesic triangle and we join the node corresponding to $\tilde{\triangle}uvw$ with the node corresponding to $\tilde{\triangle}xyz$ if they share two of their vertices (e.g., if $x = v$ and $z = w$). (See Figure 2.) As shown by Chazelle *et al.* [2], the nodes of T corresponding to the geodesic triangles whose boundaries are intersected by some ray in P will always form a path in T. Thus, if T has small diameter, then we can efficiently perform a ray-shooting query for a point p and direction $\vec{r}$ by locating the geodesic triangle whose interior contains p and then iteratively traversing geodesic triangles along direction $\vec{r}$ from p until we hit the boundary of P. Indeed, this is approach of Chazelle *et al.* for building a static ray shooting data structure.

Our approach is to maintain T as a balanced binary tree, such as a red-black tree [7, 14, 20, 25]. Sleator, Tarjan, and Thurston [24] observe that a diagonal swap between two adjacent triangles in a triangulation of a convex polygon corresponds to a rotation in the tree dual to this triangulation. We extend this to geodesic triangulations, and observe likewise that a rotation in T will correspond to swapping the diagonals determined by two adjacent geodesic triangles. We show that vertex insertion and deletion can be implemented by inserting and deleting in T, and that edge insertions and deletions can be performed using an operation on T that is analogous to a sequence of *split* and *splice* operations[1]. If we maintain geodesic paths in auxiliary structures, then we can perform each rotation and insertion in T in $O(\log n)$ time (using splits and splices on the geodesic paths involved in the rotation). We therefore achieve a running time for queries and updates that is $O(\log^2 n)$ in the worst case.

2 Preliminaries

2.1 Connected subdivisions

A *connected (planar) subdivision* $\mathcal{S}$ is a partition of the plane into simple polygons, called the *regions* of $\mathcal{S}$. Note that $\mathcal{S}$ has one unbounded region, called the external region. A subdivision $\mathcal{S}$ is generated by a planar graph embedded in the plane such that the edges are straight-line segments. We assume a standard representation for the subdivision $\mathcal{S}$, such as doubly-connected edge lists [22].

[1] Recall that in a *split*(v) operation one divides T into T_1, which contains the nodes with in-order number small than v's in-order label, and T_2, which contains the nodes with larger in-order label; a *splice*(T_1, T_2) reverses this operation.

2.2 Geodesic Triangulations

Let τ be a geodesic triangle $\tilde{\Delta}uvw$, as defined above. In general, τ will consist of a simple polygon made up of three concave chains and three piece-wise linear curves emanating away from the three vertices where the concave chains are joined. We refer to the inner polygonal region as the *deltoid* region for τ, due to its resemblance to the well-known quartic curve [17], and we refer to the three chains emanating out from the deltoid region as *tails* (see Figure 1). We represent the three concave chains using an auxiliary data structure that supports binary-type searching and chain splitting and splicing. For example, we could use red-black trees [7, 14, 20, 25].

2.3 Red-black Trees

Since our structure is built using the red-black tree data structure as a schematic, let us briefly review this structure. We use the formulation of Tarjan [25]. A *red-black tree* is a rooted binary tree T whose nodes are assigned integer ranks that obey the following constraints:

1. If v has a *nil* child pointer, then $rank(v) = 1$ and v's *nil* child pointer is viewed as pointing to a node with rank 0.
2. If v is a node with a parent, then $rank(v) \leq rank(p(v)) \leq rank(v) + 1$, where $p(v)$ denotes the parent of v.
3. If v is a node with a grandparent, then $rank(v) < rank(p(p(v)))$.

A node v is called *black* if $rank(p(v)) = rank(v) + 1$ or v is the root; v is *red* otherwise (i.e., if $rank(p(v)) = rank(v)$).

As a shorthand notation, let us use $rank(T)$ to denote the rank of the root of a tree T. Let n be the number of nodes of T. It is easy to see that $rank(v)$ is proportional to the logarithm of the number of descendents of v, so that $rank(T) = O(\log n)$. Tarjan [25] shows that red-black trees support the *split* and *splice* operations in $O(r(n) \log n)$ time, where $r(n)$ denotes the time complexity of performing a rotation[2] in T. His methods are based on using rotations and simple node and edge insertions and deletions. In our use of red-black trees, we must assume that each internal node has degree 3; thus, let us assume that the root of a red-black tree T actually has a parent, which is a degree-one "dummy node." In addition, we desire that our tree-modification operations be based strictly on the use of tree rotations, and not use the more general pointer changing as is used in the standard implementations [7, 14, 20, 25]. Fortunately, such implementations are easy to come by. For completeness, we include an outline here.

For a *splice* of trees T_1 and T_2, we create a new node r (if one doesn't already exist) such that the root of T_1 and the root of T_2 form the children of r. We then perform a series of rotations in this tree to push the smaller tree down to an appropriate depth. The time needed is $O(r(n)|rank(T_1) - rank(T_2)|) = O(r(n) \log n)$.

Likewise, let us describe a *non-destructive* version of a *split* of tree T at a node $v \in T$ that returns a node r whose left child is a red-black tree for the elements left of v in T, and whose right child is a node s, where s's left child is v and s's right child is a red-black tree for the elements right of v in T. Such a tree can be constructed from T by performing a series of rotations to move v up T. Any time a pair of nodes on the left fringe[3] (resp., right fringe) of the path from the root of T to v become siblings during this series of rotations, we perform a *splice* of their respective subtrees. (See Figure 3.) The final tree can be made to produce the result described above. One can show that the time needed to perform this operation is $O(r(n) \log n)$, for the total cost of performing all the splice operations forms a "telescoping sum" that is $O(r(n) \log n)$.

Finally, we must contend with the fact that the root of our red-black tree implementations has a "dummy node" parent. In particular, we allow for one to perform an *evert*(v) operation on a red-black tree T, where one makes a leaf node v be the new "dummy node" parent of the root, and lets the old dummy node become a

[2] In the standard red-black tree setting $r(n)$ is $O(1)$, but this will not be the case in our applications.

[3] Recall that a *left* (resp., *right*) fringe node for a leaf-to-root path π is a node that is a left (resp., right) child of a node on π but is itself not on π.

regular leaf node. Of course, this also requires that we rebalance T. Such a rebalancing can be implemented by prefacing the eversion by performing a non-destructive split at the leaf node v, which divides T into T_1, which contains the nodes to the left of v, and T_2, which contains the nodes to the right of v. Then we may perform the *evert* operation, and rebalance the tree by splicing together T_1 and the old dummy node for the root of T, and then splicing the resulting tree with T_2. Since this requires $O(1)$ *split* and *splice* operations, it clearly can be implemented in $O(r(n) \log n)$ time.

3 Our Data Structure

Let $\mathcal{S}$ be a connected subdivision. In this section we describe our data structure for performing ray shooting queries in $\mathcal{S}$.

3.1 The primary structure

As mentioned in the introduction, the main component of our data structure for $\mathcal{S}$ is that we maintain a geodesic triangulation of each region of $\mathcal{S}$. With each region P of $\mathcal{S}$, we also store the tree T dual to the geodesic triangulation we maintain for P. Each internal node in T corresponds to a geodesic triangle and we join the node corresponding to $\tilde{\Delta} uvw$ with the node corresponding to $\tilde{\Delta} xyz$ if they share two of their vertices (e.g., if $x = v$ and $z = w$). Each leaf corresponds to an edge of P and is joined to the (parent) geodesic triangle that has this edge on its boundary. In particular, if one of the edges of a geodesic triangle τ is also an edge of P, then we say that τ is a *border* triangle, and, for each such border triangle τ, we add an adjacency in T from the node associated with τ to a (leaf) node associated with the edge of P on τ (see Figure 5). In addition, we distinguish a border triangle ρ in P as the *root* triangle, so as to root T at the node associated with ρ. The main idea of our primary structure, then, is to maintain this rooted tree T as a red-black tree [7, 14, 20, 25], ignoring the (dummy) leaf node associated with ρ.

3.2 The secondary point location structure

As a secondary data structure we maintain a dynamic point location data structure on the deltoid regions determined by the geodesic triangulations of all the faces in $\mathcal{S}$. In particular, we use the structure of Goodrich and Tamassia [11], which uses $O(n)$ space, supports point location queries in $O(\log^2 n)$ time, edge insertion and deletion in $O(\log n)$ time, and vertex insertion and deletion in $O(\log n)$ time as well. The only caveat to using this structure is that it requires each face in the subdivision to be monotone (say, with respect to the x-axis). That is, it requires the underlying subdivision to be *monotone*. Of course, a deltoid region need not be monotone. Nevertheless,

Lemma 3.1: *The geodesic triangulation of a connected subdivision can be refined to a monotone subdivision by inserting at most one edge in each deltoid region.*

Proof: Omitted in this preliminary version. □

Thus, our secondary structure consists of the dynamic point location of Goodrich and Tamassia [11] built upon the union of the deltoid regions in all the geodesic triangles in $\mathcal{S}$, together with at most one edge per deltoid region so as to make each face in the resulting subdivision $\mathcal{S}'$ monotone with respect to the x-axis.

3.3 The tertiary deltoid structures

The final component of our data structure is a tertiary structure built for the deltoid regions. In particular, for each deltoid region δ, we maintain each of the three concave chains for δ in a balanced tree structure (e.g., a red-black tree [7, 14, 20, 25]). Each internal node in such a tree corresponds to a subchain of a concave chain and stores the length of the associated subchain.

Our entire data structure, $\mathcal{D}$, then, consists of the primary geodesic triangulation structures, the secondary point location structure, and the tertiary deltoid structures.

Lemma 3.2: *$\mathcal{D}$ requires $O(n)$ space.*

Proof: Omitted in this preliminary version. □

3.4 Ray shooting

So, suppose we have such a data structure for our connected subdivision $\mathcal{S}$, and let $\vec{r}$ be a query ray for which we wish to perform a ray shooting query. We begin by performing a point location for the origin of $\vec{r}$ using the secondary point location structure. This takes time $O(\log^2 n)$ [11]. We then traverse the geodesic triangulation along the ray $\vec{r}$ one triangle at a time, until the region boundary is hit. For each geodesic triangle traversed, we perform a stabbing query for $\vec{r}$ and the triangle boundary to identify $\vec{r}$'s exit point using the tertiary structures stored for each geodesic triangle. (See Figure 4.) Since we maintain T as a red-black tree, $O(\log n)$ triangles are traversed, each of which requires $O(\log n)$ time for its stabbing query. Therefore, we have

Lemma 3.3: *A ray-shooting query in $\mathcal{D}$ takes $O(\log^2 n)$ time.*

4 Balanced Geodesic Triangulations in a Dynamic Environment

In this section we show how to maintain the data structure $\mathcal{D}$ while performing edge insertion and deletion as well as vertex insertion and deletion. In particular, we define the following update operations on a connected subdivision $\mathcal{S}$:

InsertEdge$(e, v, w, R; R_1, R_2)$: Insert edge $e = (v, w)$ into region R such that R is partitioned into two regions R_1 and R_2.

RemoveEdge$(e, v, w, R_1, R_2; R)$: Remove edge $e = (v, w)$ and merge the regions R_1 and R_2 formerly on the two sides of e into a new region R.

InsertVertex$(v, e; e_1, e_2)$: Split the edge $e = (u, w)$ into two edges $e_1 = (u, v)$ and $e_2 = (v, w)$ by inserting vertex v along e.

RemoveVertex$(v, e_1, e_2; e)$: Let v be a vertex with degree two such that its incident edges $e_1 = (u, v)$ and $e_2 = (v, w)$, are on the same straight line. Remove v and merge e_1 and e_2 into a single edge $e = (u, w)$.

AttachVertex$(v, e; w)$: Insert edge $e = (v, w)$ and degree-one vertex w inside some region R, where v is a vertex of R.

DetachVertex(v, e): Remove a degree-one vertex v and edge e incident on v.

The above repertory of operations is complete for connected subdivisions. That is, any connected subdivision $\mathcal{S}$ can be constructed "from scratch" using only the above operations. Also, *AttachVertex* and *DetachVertex* can be simulated by a ray shooting query followed by a sequence of $O(1)$ *InsertVertex*, *RemoveVertex*, *InsertEdge*, and *RemoveEdge* operations [4]. Hence, such operations will not be further discussed.

4.1 Rotations

A swap of diagonals in the geodesic triangulation of a region corresponds to a rotation in the dual tree (see Figure 5). The geodesic triangulation is modified with $O(1)$ *InsertEdge*/*RemoveEdge* operations. The boundaries of the geodesic triangles are modified by $O(1)$ split/splice operations (see Figure 6). Thus, a rotation requires logarithmic time, i.e., $r(n)$ is $O(\log n)$ in our primary structure T.

4.2 Vertex Insertion and Deletion

Operations *InsertVertex*$(v, e; e_1, e_2)$ and *RemoveVertex*$(v, e_1, e_2; e)$ correspond to the insertion/deletion of a node in the dual trees associated with the regions that share edge e. The geodesic triangulation is modified by two *InsertVertex*/*RemoveVertex* operations. The boundaries of the geodesic triangles are modified by two insertions/deletions.

Lemma 4.1: *Operations InsertVertex and RemoveVertex take each $O(\log n)$ time.*

4.3 Edge Insertion and Deletion

Let us next consider edge insertion and deletion, and begin our discussion with the insertion case. The operation *InsertEdge*$(e, v, w, R; R_1, R_2)$ can be implemented as follows. Let d and f be edges of R such that d is incident to v and f is incident to w, with d and f being on opposite sides of e (i.e., d and f will be separated after e is inserted). We begin our implementation of the insertion of e by everting the tree T at the leaf for d, resulting in a geodesic triangulation of R corresponding to a red-black tree T' rooted at d. We then perform a non-destructive split on the dual tree T' at f so that the edge e is the diagonal between the geodesic triangles corresponding to the parent and grandparent of d, which gives us a new dual tree T''. We may then insert the edge e, cutting T'' at the edge dual to e. This results in two new regions R_1 and R_2 with corresponding dual trees T_1 and T_2. Notice that the root of T_1 (resp., T_2) has as one of its children the root of a red-black tree and as its other child the node d (resp., f). We complete the construction, then, by performing a *splice* on the two children of the root of T_1 and the root of T_2, respectively. (See Figure 7.) Note that this construction requires that we perform $O(1)$ *evert*, *split*, and *splice* operations on the dual trees for R_1 and R_2. Each red-black tree rotation required to implement these operations in T requires $O(\log n)$ time using the tertiary chain structures. Thus, this edge insertion can be implemented in $O(\log^2 n)$ time.

Let us therefore next consider the operation *RemoveEdge*$(e, v, w, R_1, R_2; R)$. Let T_1 and T_2 be the dual trees for the geodesic triangulations of R_1 and R_2, respectively. We begin by performing an *evert* operation on T_2 to make the leaf corresponding to e become the root for this new tree T_2' in R_2. We then perform a non-destructive *split* on T_1 at the leaf in T_1 corresponding to e, which gives us a new tree T_1'. We then conceptually merge R_1 and R_2 by replacing the leaf for e in T_1' with the root of T_2'. That is, if we let r denote the root of T_2', then we replace the leaf for e by r. We complete the construction by performing a *splice* at the (new) parent for r, and then another *splice* at the grandparent of r. (See Figure 8.) This gives us a balanced tree for the entire region R. Notice that the implementation of this operation required $O(1)$ *evert*, *split*, and *splice* operations. Thus, it too can be implemented in $O(\log^2 n)$ time.

Lemma 4.2: *Operations InsertEdge and RemoveEdge take each* $O(\log^2 n)$ *time.*

5 Shortest Path Queries

In this section, we show how to extend our approach so as to efficiently answer shortest path queries in $\mathcal{S}$. In this case we are given two query point p and q and we wish to determine the shortest path between p and q that does not cross any edges of $\mathcal{S}$. We may assume, without loss of generality, that p and q belong to the same region in $\mathcal{S}$, since we can test if this is not the case in $O(\log^2 n)$ time [11]. So, suppose we are given two query points p and q in a region P of $\mathcal{S}$, and we wish to perform a shortest path query for the pair (p, q). We consider two variations of this query: reporting the length of the path, and reporting all the edges of the path.

In the following, an augmented balanced binary tree, called *chain tree*, will be used to represent a polygonal chain, where the leaves are associated with the edges, and the internal nodes with the vertices of the chain. Each node also corresponds to a subchain and stores its length. It should be clear that this information can be updated in $O(1)$ time per rotation, so that splitting or splicing two chain trees takes logarithmic time. With this representation, it is possible to find the two tangents from a point to a convex chain and the four common tangents between two convex chains in logarithmic time [22].

In order to support shortest path queries, we extend our data structure so as to store entire geodesic triangles. Specifically, we modify our data structure by storing at each node μ of tree T the tails of the geodesic triangle τ associated with μ, in addition to a representation of the deltoid region for τ. In order to maintain this as a linear-space structure we do not store any portions already stored at an ancestor of μ, however. The portions of tails stored at a node are represented with chain trees, and the missing chains are represented in $O(1)$ space as a pair v, w representing the interval of chain edges from v to w (which is a shortest path from v to w stored at an ancestor), where v and w are vertices of the geodesic triangle for μ.

Lemma 5.1: *The space requirement of the modified data structure is $O(n)$; the portion of the geodesic triangle stored at a node consists of $O(1)$ chains; and rotations in T can be performed in $O(\log n)$ time.*

Proof: We omit the details in this preliminary version. □

If p and q are vertices of R, the geodesic path algorithm is as follows: First, we evert T so that the dummy leaf of T is associated with an edge incident on p. This takes $O(\log^2 n)$ time. Next, perform a non-destructive split at a leaf μ_q of T incident upon q to bring μ_q to be the grandchild of the root of T so that the geodesic path from p to q is the diagonal separating the geodesic triangle for μ_q from the geodesic triangle for $p(\mu_q)$ (see Figure 7, as this is very similar to our operation for edge insertion with $v = p$ and $w = q$). Now, the shortest path between p and q is a diagonal in the geodesic triangulation for R, so that the length of the geodesic path and its k edges can be retrieved in time $O(1)$ and $O(k)$, respectively, from the chain trees. Finally, we undo the above rotations to reset the data structure to its original state. The overall time complexity is $O(\log^2 n)$, plus $O(k)$ if the path is reported in addition to its length.

If p and q are not vertices of R, we "attach" them to the boundary of R by means of two horizontal ray-shootings followed by two *AttachVertex* operations, which takes $O(\log^2 n)$ time, and we apply the previous method.

Lemma 5.2: *A shortest-path query takes time $O(\log^2 n)$ to report the length of the path, plus $O(k)$ time to report the k edges of the path.*

6 Conclusion

We have given a simple and efficient scheme for dynamically maintaining a connected subdivision $\mathcal{S}$ subject to ray shooting and shortest path queries. Our method was based on maintain geodesic triangulations of each polygonal region in $\mathcal{S}$ through the use of an elegant duality between diagonal swaps between adjacent geodesic triangles and rotations in red-black trees. Since we implemented each rotation in $O(\log n)$ time, this resulted in worst-case running times of $O(\log^2 n)$ for queries and updates.

Hershberger and Suri [16] recently showed that one can triangulate the interior of a simple polygon (using additional interior points) so that any ray intersects $O(\log n)$ triangles. Applying our approach to this method would not improve the running time of updates, however, since an edge insertion would still require changing $O(\log n)$ edges, and we would still require a dynamic point location structure. Thus, this would still require $O(\log^2 n)$ time. Therefore, this still leaves as an open question whether one can achieve $o(\log^2 n)$ time for both updates and ray shooting queries in a dynamic connected subdivision.

Acknowledgement

We would like to thank Günter Rote for useful comments.

References

[1] P.K. Agarwal and M. Sharir, "Applications of a new partitioning scheme," *Proc. 2nd Workshop Algorithms Data Struct.*, Lecture Notes in Computer Science, vol. 519, Springer-Verlag, 1991, 379–391.

[2] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink, "Ray shooting in polygons using geodesic triangulations," *Proc. Int. Coll. on Automata, Languages, and Programming (ICALP): LNCS 510*, 1991, 661–673.

[3] B. Chazelle and L.J. Guibas, "Visibility and Intersection Problems in Plane Geometry," *Discrete Comput. Geom.*, **4**, 1989, 551–581.

[4] Y.-J. Chiang, Y.-J., F.P. Preparata, and R. Tamassia, "A Unified Approach to Dynamic Point Location, Ray Shooting, and

Shortest Paths in Planar Maps," *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, 1993, 44–53.

[5] S.W. Cheng and R. Janardan, "New Results on Dynamic Planar Point Location," Technical Report TR 90-13, Dept. of Computer Science, Univ. of Minnesota, 1990. (Prelim. version: *31st FOCS*, 96–105, 1990.)

[6] S.W. Cheng and R. Janardan, "Space-efficient ray shooting and intersection searching: algorithms, dynamization and applications," *Proc. 2nd ACM-SIAM Sympos. Discrete Algorithms*, 1991, 7–16.

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press (Cambridge, Mass.: 1990).

[8] Eppstein, D., G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook, and M. Yung, "Maintenance of a Minimum Spanning Forest in a Dynamic Planar Graph," *Proc. First ACM-SIAM Symp. on Discrete Algorithms*, 1–11, 1990.

[9] O. Fries, K. Mehlhorn, and S. Naeher, "Dynamization of Geometric Data Structures," *Proc. (1st) ACM Symp. on Computational Geometry*, 168–176, 1985.

[10] M.T. Goodrich, M. Ghouse, and J. Bright, "Generalized Sweep Methods for Parallel Computational Geometry," *Proc. 2nd ACM Symp. on Parallel Algorithms and Architectures*, 1990, 280–289.

[11] M.T. Goodrich and R. Tamassia, "Dynamic Trees and Dynamic Point Location," *Proc. 23rd ACM Symp. on Theory of Computing*, 1991, 523–533.

[12] L.J. Guibas and J. Hershberger, "Optimal shortest path queries in a simple polygon," *J. Comput. Syst. Sci.*, **39**, 1989, 126–152.

[13] L.J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R.E. Tarjan, "Linear Time Algorithms for Visibility and Shortest Path Problems Inside Simple Polygons," *2nd ACM Comp. Geom.*, 1986, 1–13.

[14] L.J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proc. 19th IEEE Symp. on Foundations of Computer Science*, 1978, 8–21.

[15] J. Hershberger, "Optimal Parallel Algorithms for Triangulated Simple Polygons," *Proc. 8th ACM Symp. on Computational Geometry*, 1992, 33–42.

[16] J. Hershberger and S. Suri, "A Pedestrian Approach to Ray Shooting: Shoot a Ray, Take a Walk," *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, 1993.

[17] J.D. Lawrence, *A Catalog of Special Plane Curves*, Dover Publications, Inc. (New York: 1972).

[18] D.T. Lee and F.P. Preparata, "Euclidean shortest paths in the presence of rectilinear barriers," *Networks*, **14**, 1984, 393–410.

[19] E.M. McCreight, "Priority Search Trees," *SIAM J. on Comput.*, No. 14, 1985, 257–276.

[20] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, 1984.

[21] M. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, Springer-Verlag, 1983.

[22] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, NY, 1985.

[23] D.D. Sleator and R.E. Tarjan, "A Data Structure for Dynamic Trees," *J. Comput. and Sys. Sci.*, Vol. 26, 362–391, 1983.

[24] D.D. Sleator, R.E. Tarjan, and W. P. Thurston, "Rotation distance, triangulations, and hyperbolic geometry," *J. Amer. Math. Soc.*, **1**, 1988, 647–682.

[25] R.E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.

[26] D. Willard and G. Lueker, "Adding Range Restriction Capability to Dynamic Data Structures," *J. ACM*, Vol. 32, 597–617, 1985.

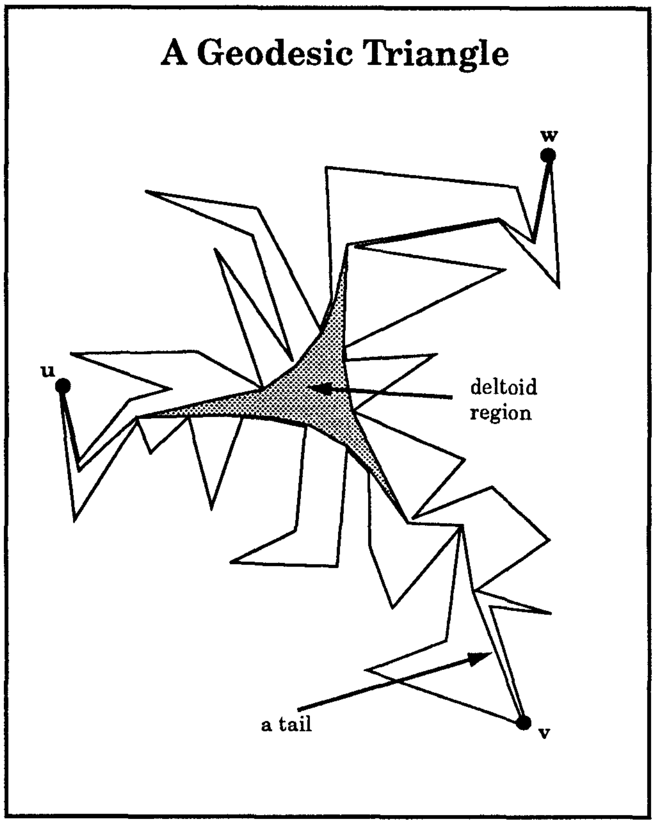

Figure 1

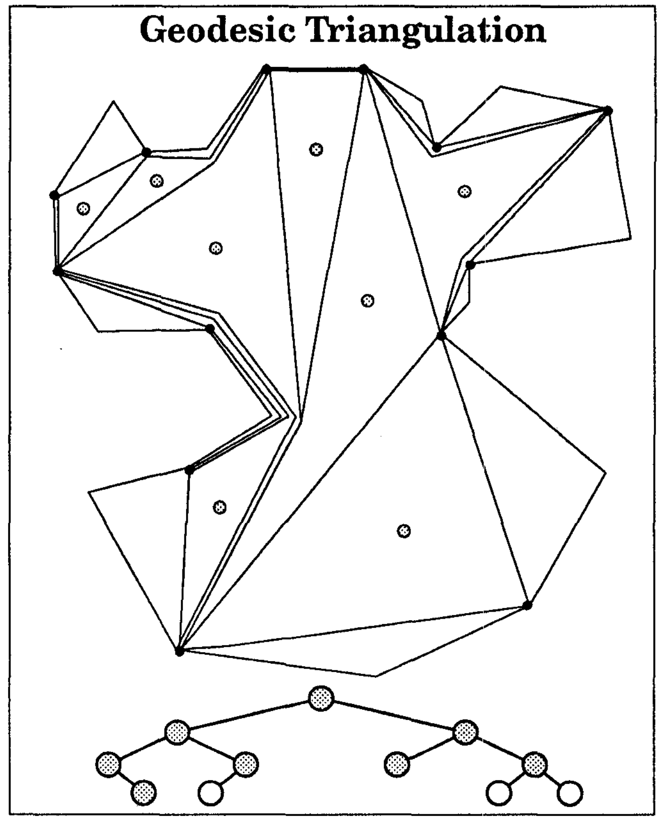

Figure 2

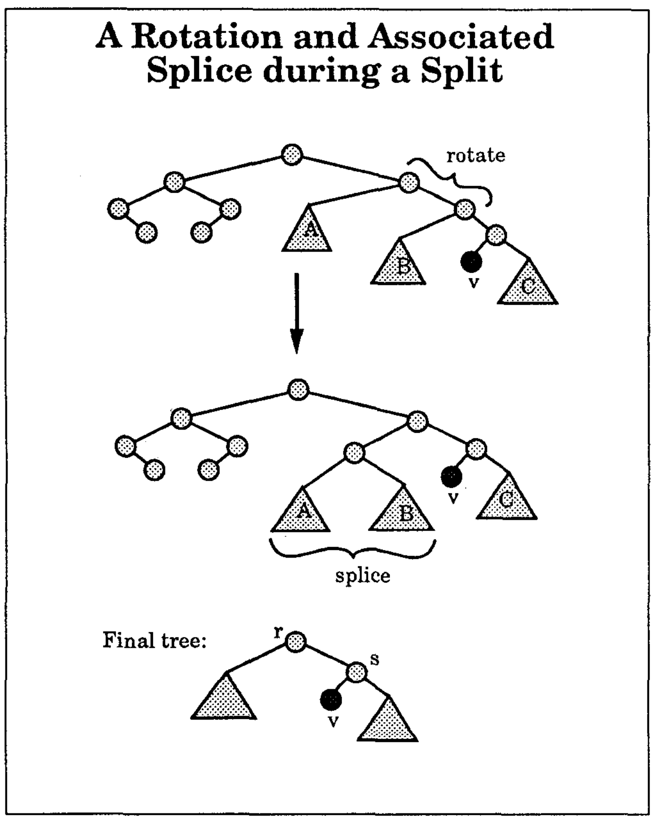

Figure 3

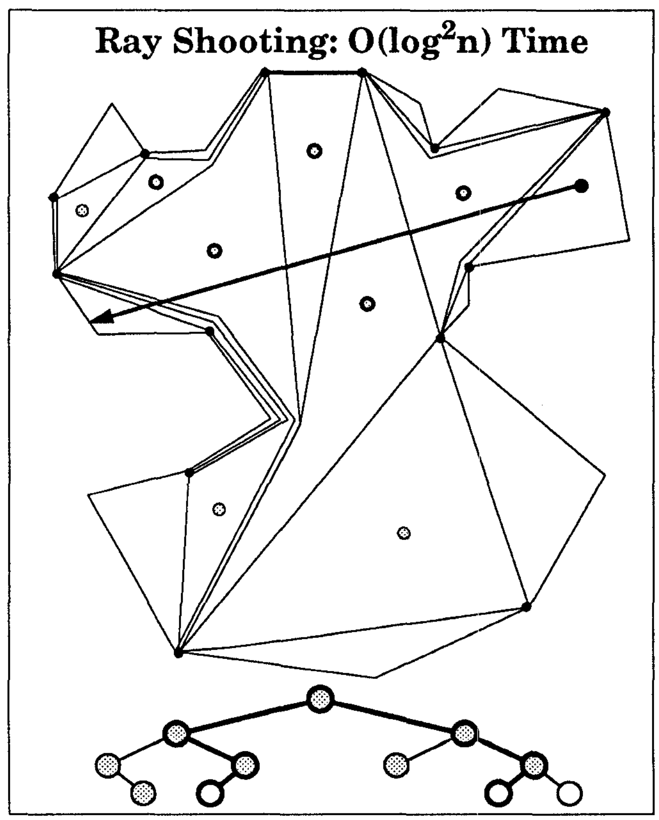

Figure 4

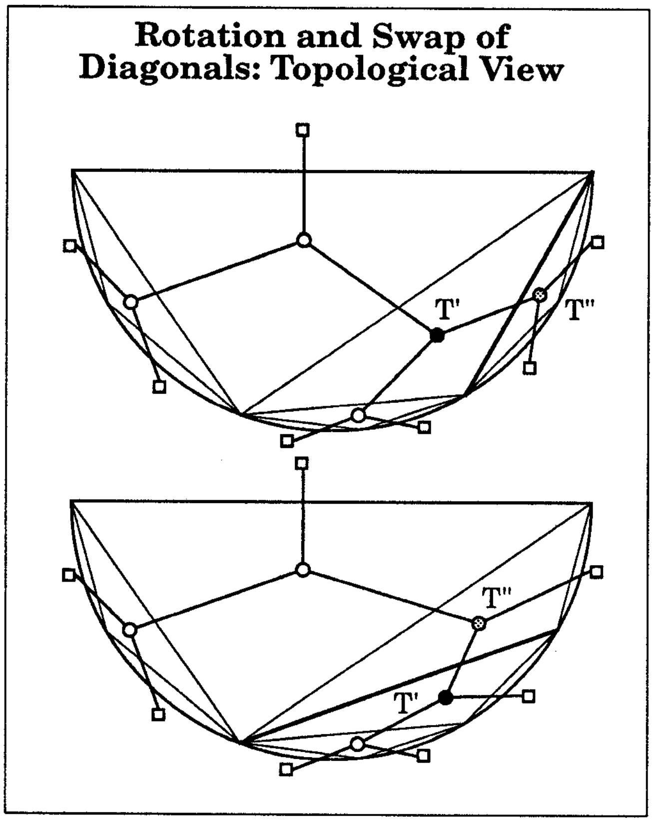

Figure 5

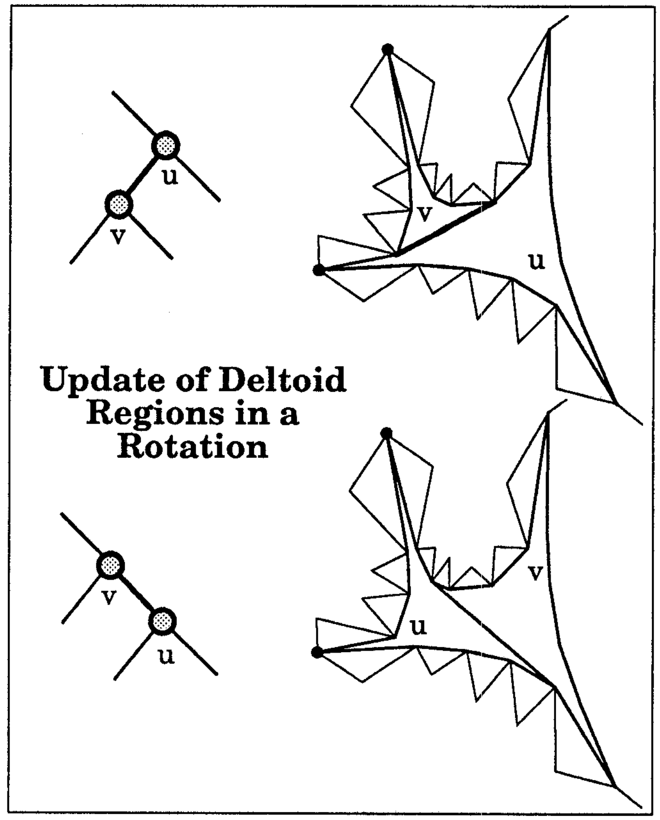

Figure 6

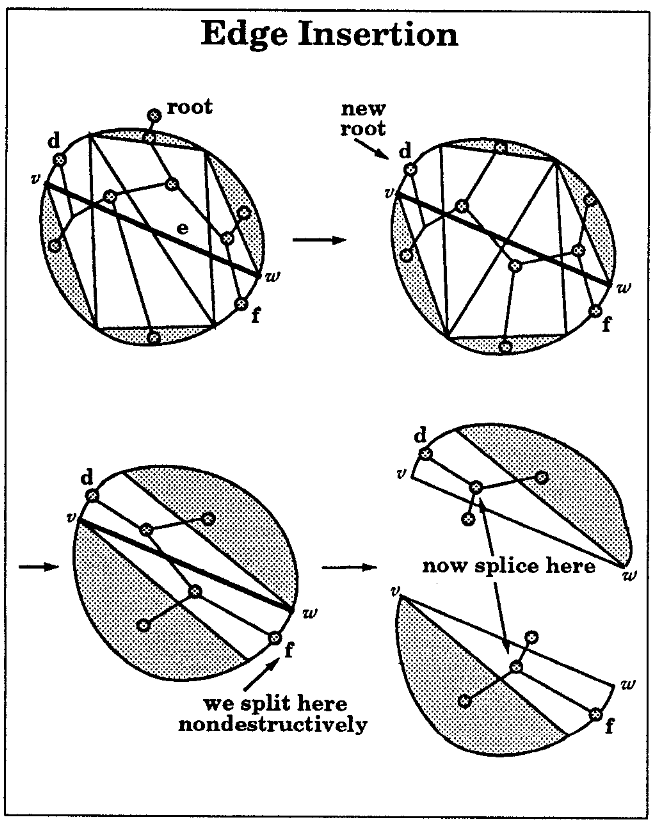

Figure 7

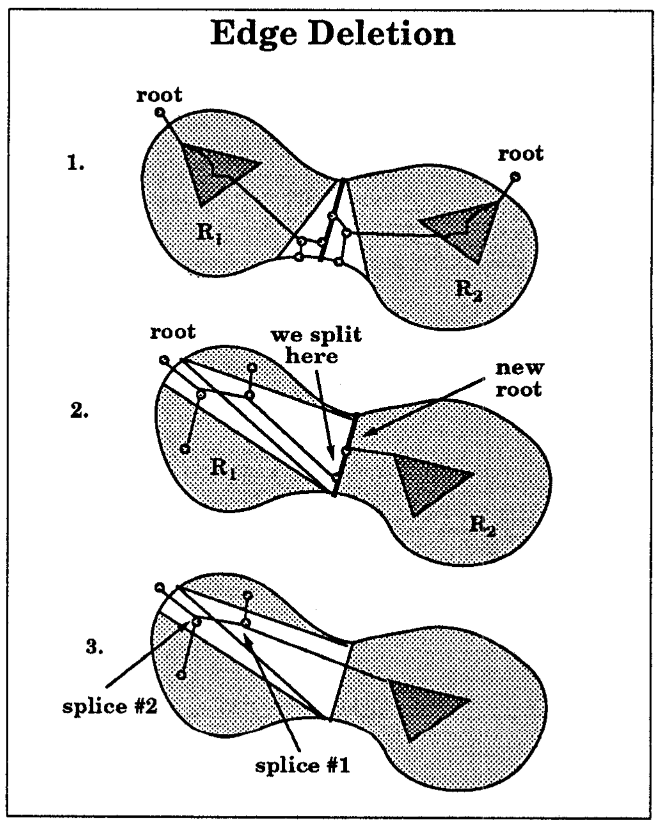

Figure 8