Algorithmica (1994) 12: 54–68

Algorithmica

Ray Shooting in Polygons Using Geodesic Triangulations[1]

B. Chazelle,[2] H. Edelsbrunner,[3] M. Grigni,[4] L. Guibas,[5,6,7]
J. Hershberger,[5] M. Sharir,[8,9] and J. Snoeyink[7]

Abstract. Let $\mathcal{P}$ be a simple polygon with n vertices. We present a simple decomposition scheme that partitions the interior of $\mathcal{P}$ into $O(n)$ so-called geodesic triangles, so that any line segment interior to $\mathcal{P}$ crosses at most $2 \log n$ of these triangles. This decomposition can be used to preprocess $\mathcal{P}$ in a very simple manner, so that any ray-shooting query can be answered in time $O(\log n)$. The data structure requires $O(n)$ storage and $O(n \log n)$ preprocessing time. By using more sophisticated techniques, we can reduce the preprocessing time to $O(n)$. We also extend our general technique to the case of ray shooting amidst k polygonal obstacles with a total of n edges, so that a query can be answered in $O(\sqrt{k} \log n)$ time.

Key Words. Computational geometry, Ray-shooting, Triangulation.

1. Introduction. In this paper we consider the *ray-shooting* problem in simple polygons. Given a simple polygon $\mathcal{P}$ with n vertices, we wish to preprocess it into a data structure that supports fast ray-shooting queries inside $\mathcal{P}$, each asking for the first intersection of a query ray with the polygon. This is one of the fundamental problems in plane computational geometry. It has been studied by Chazelle and Guibas [7], who gave a solution with preprocessing $O(n \log n)$, and optimal query time $O(\log n)$ and storage $O(n)$. Unfortunately, this technique is exceedingly involved and requires sophisticated data structures that are accessed in a complex manner. To the best of our knowledge this is still the only known technique that achieves (asymptotically) optimal query time and storage.

In this paper we present two variants of an alternative solution to the ray-shooting problem. The simpler variant takes $O(\log^2 n)$ time for a query, and besides

[1] Work by Bernard Chazelle has been supported by NSF Grant CCR-87-00917. Work by Herbert Edelsbrunner has been supported by NSF Grant CCR-89-21421. Work by Micha Sharir has been supported by ONR Grants N00014-89-J-3042 and N00014-90-J-1284, by NSF Grant CCR-89-01484, and by grants from the U.S.–Israeli Binational Science Foundation, the Fund for Basic Research administered by the Israeli Academy of Sciences, and the G.I.F., the German–Israeli Foundation for Scientific Research and Development.
[2] Computer Science Department, Princeton University, Princeton, NJ 08544, USA.
[3] Computer Science Department, University of Illinois, Urbana, IL 61801, USA.
[4] Department of Mathematics, MIT, Cambridge, MA 02139, USA.
[5] DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA.
[6] Laboratory for Computer Science, MIT, Cambridge, MA 02139, USA.
[7] Computer Science Department, Stanford University, Stanford, CA 94305-4055, USA.
[8] School of Mathematical Sciences, Tel Aviv University, Ramat Aviv 69 978, Israel.
[9] Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA.

Received September 17, 1991; revised August 22, 1992. Communicated by Kurt Mehlhorn.

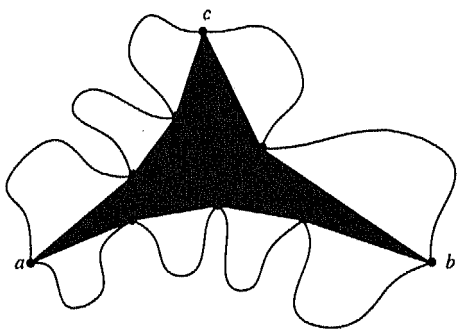

Fig. 1. A geodesic triangle.

performing point location, its most complex operation is binary search. The second variant improves on the first one by the use of weight-balanced trees and fractional cascading, thus obtaining query time $O(\log n)$. Both variants require storage $O(n)$ and preprocessing time $O(n \log n)$. The second variant is thus optimal in terms of query time and storage. It is much simpler than the technique of [7] and should be easy to implement. By using the hourglass technique of [11], and the recent linear-time triangulation algorithm of [3], we can reduce the preprocessing cost to $O(n)$, thus achieving full optimality.

Our technique is based on a certain triangulation-like decomposition of the interior of $\mathcal{P}$ into regions that we call *geodesic triangles*. Such a triangle is formed by three shortest paths inside $\mathcal{P}$, that connect three vertices, a, b, c, of $\mathcal{P}$, so that all three paths are concave (see Figure 1). Our decomposition, referred to as *balanced geodesic triangulation*, has the property that any line segment interior to $\mathcal{P}$ crosses only $O(\log n)$ of the geodesic triangles. The strategy of our solution is now obvious: given a query ray, locate its starting point in the decomposition and traverse the geodesic triangles by walking along the ray until it hits the boundary of $\mathcal{P}$. By repeated binary search, the walk can be done in time $O(\log^2 n)$. Weight-balanced trees and fractional cascading can be used to improve the running time to $O(\log n)$.

We believe that the balanced geodesic deomposition of a simple polygon is of independent interest and is likely to find other applications. For example, it can be used to show that for every simple n-gon there is a triangulation into $O(n)$ triangles (using Steiner points), so that any interior line segment meets at most $O(\log^2 n)$ triangles. We also generalize the geodesic triangulation and our ray-shooting solutions to the case of k polygonal obstacles with a total of n edges. In this problem a query takes time $O(\sqrt{k} \log n)$, which is an improvement of the best previous solution in [1] by a factor of $\log n$.

The paper is organized as follows. Section 2 presents the balanced geodesic triangulation and derives our first simple ray-shooting algorithm. Section 3 explains how to construct a geodesic triangulation in linear time. Section 4 shows how a balanced geodesic triangulation can be used to obtain a real triangulation that satisfies the condition mentioned above. Section 5 describes the refined

ray-shooting solution, and Section 6 extends the technique to the case of polygonal obstacles.

2. The Balanced Geodesic Triangulation. We begin our discussion with a remarkably simple and practical ray-shooting algorithm. It allows us to shoot a ray in a simple n-gon in $O(\log^2 n)$ time, using $O(n \log n)$ preprocessing and $O(n)$ storage. The main tool is a particular decomposition of the polygon $\mathcal{P}$ into geodesic triangles, which has the property that no line segment interior to $\mathcal{P}$ can cut more than a logarithmic number of these triangles. We call such a decomposition a *balanced geodesic triangulation.* Recall that the *geodesic path* between two vertices of the polygon is the shortest path that connects the vertices and stays completely within the polygon. We construct a balanced geodesic triangulation by drawing collections of geodesic paths in roughly $\log n$ stages, as described below.

Before we give a formal description it is helpful to visualize the decomposition by the following intuitive process. Deform the polygon into a regular convex polygon by moving, shrinking, or stretching the edges. Then add all the chords shown in Figure 2, thereby building a hierarchy similar to the two-dimensional hierarchy of Dobkin and Kirkpatrick [8]. Think of the chords as tight rubber bands and now deform the polygon back into its original shape. The resulting decomposition of $\mathcal{P}$ is exactly what we want.

Algorithmically, the desired decomposition is built in a logarithmic number of recursive stages. In the first stage we choose three vertices of the polygon that are equally spaced around the boundary of the polygon and we connect them with geodesic paths. Specifically, if $v_1, v_2, \ldots, v_n$ are the vertices of $\mathcal{P}$, we compute

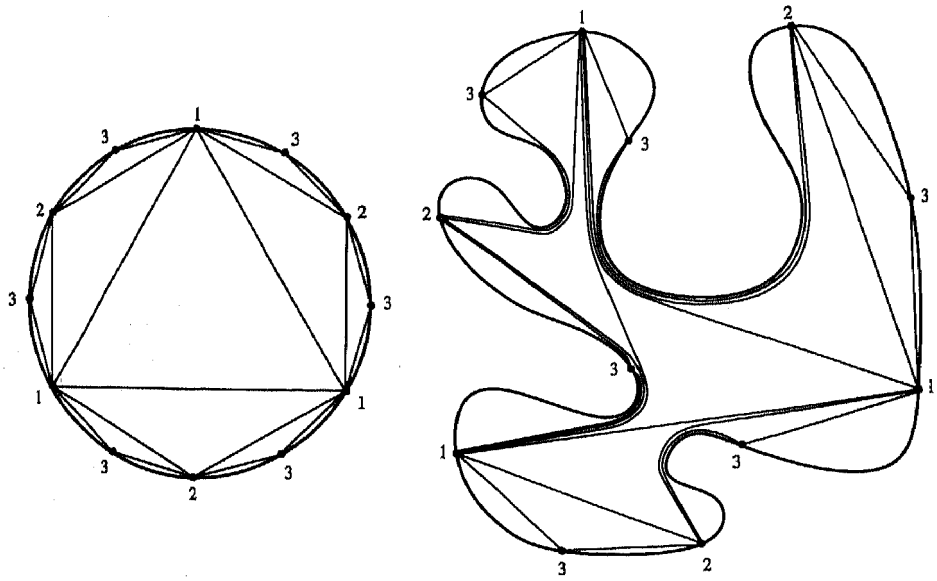

Fig. 2. The rubber-band view of a balanced geodesic triangulation.

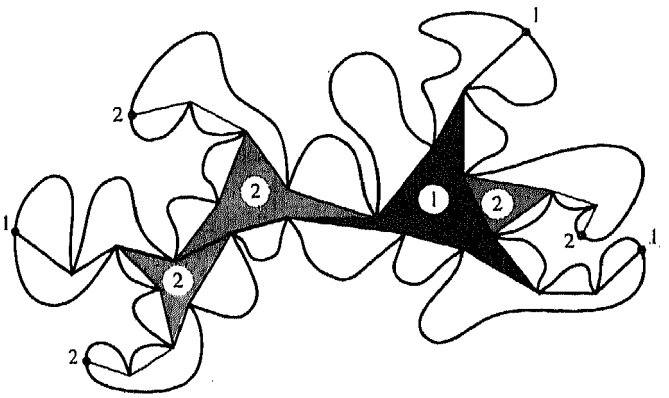

Fig. 3. The first two levels of a balanced geodesic triangulation.

the geodesic paths connecting v_1 and $v_{\lfloor n/3 \rfloor}$, $v_{\lfloor n/3 \rfloor}$, and $v_{\lfloor 2n/3 \rfloor}$, as well as v_1 and $v_{\lfloor 2n/3 \rfloor}$. In the second stage we repeat the same operation but now we connect the pairs $(v_1, v_{\lfloor n/6 \rfloor})$, $(v_{\lfloor n/6 \rfloor}, v_{\lfloor n/3 \rfloor})$, $(v_{\lfloor n/3 \rfloor}, v_{\lfloor n/2 \rfloor})$, etc. We iterate on this process until the geodesic paths connect pairs of vertices that are only one vertex apart. Figure 3 gives a detailed view of the construction after the second stage.

Let us take a closer look at how these shortest paths partition $\mathcal{P}$ into regions. Let a, b be two vertices connected by a geodesic path at stage k and let c be the middle vertex that gets connected to a and b at the next stage. The two paths connecting each of the pairs (a, c) and (b, c) at stage $k + 1$, together with the path connecting (a, b) at stage k, form a figure called a *kite*. It consists of a central *geodesic triangle* (the interior of the kite) and three (possibly empty) paths connecting the vertices of the "triangle" with the vertices a, b, c. The geodesic triangle consists of three (disjoint) concave portions of the original paths. Such a kite is also formed between the three initial paths drawn in the first stage. It should be noted that it is possible for a kite to have an empty geodesic triangle; this happens when the concatenation of two of its paths form the third path. For example, think of a boomerang-like polygon consisting of a V-shaped two-edge path connected by a concave polygonal path to which the three vertices chosen at stage k belong.

Each kite in the decomposition appears at a unique stage k. Except at the first stage ($k = 1$), one of the sides of a kite is naturally associated with stage $k - 1$ while the two others are associated with stage k. The kth stage in the construction gives rise to at most $3 \cdot 2^{k-1}$ geodesic paths, most of which are likely to share common edges with paths previously drawn. The number of kites generated at the kth stage is 1 for $k = 1$ and at most $3 \cdot 2^{k-2}$ for $k > 1$.

The decomposition process can be modeled by a free degree-3 tree of diameter at most $2 \log n$, which we call the *decomposition tree*. Each internal node is associated with a distinct kite, but later we keep only the corresponding geodesic triangle (if nonempty), to avoid storing an edge more than twice. Since the decomposition process is recursive, it is easy to prove by induction on the number

of vertices that a balanced geodesic triangulation of $\mathscr{P}$, indeed, forms a partition of the polygon into geodesic triangles. No new vertices are added, and therefore the total number of distinct edges introduced in this triangulation is at most $n - 3$. We claim that no line segment s interior to $\mathscr{P}$ can intersect more than a logarithmic number of edges. To see this, note that the sequence of kites intersected by s is a simple path in the decomposition tree (since a kite that we leave can never be re-entered). The claim follows because the tree has diameter at most $2 \log n$.

We now show how to compute a balanced geodesic triangulation of $\mathscr{P}$ in $O(n \log n)$ time. First we triangulate the polygon in time $O(n \log n)$ [21]. To compute a geodesic path between two points it now suffices to navigate from one point to the other by crossing triangles in an incremental fashion, while maintaining a funnel structure, as explained in [2] and [17]. The cost of the computation is proportional to the number of triangles crossed by the geodesic path. In other words, the time to compute all the geodesic paths at all stages is equal, within a constant factor, to the total number of intersections between the edges of the geodesic paths and the edges of the triangulation. We just saw that a given segment cannot intersect more than a logarithmic number of geodesic edges, so the preprocessing time is $O(n \log n)$.

Given a query ray (q, α), where q is a point in $\mathscr{P}$ and α is the shooting direction, what is the first point of the boundary of $\mathscr{P}$ to be hit by the ray? As it turns out, the hierarchical nature of the balanced geodesic triangulation is a useful conceptual device to analyze the complexity of the algorithms operating on the structure, but it is not needed by the algorithms *per se*. Thus, to answer the query we begin by finding which geodesic triangle Δ contains q, using any one of the optimal point-location methods known to date [9], [16]. (Note that this adds only a linear term to the preprocessing time.) Then we explore Δ to find which edge is hit by the query. We can do this in $O(\log n)$ time by observing that the boundary of Δ consists of three concave chains, each of which can be tested for intersection with a line by using binary search [21]. If the edge hit by the ray is an edge of the boundary of $\mathscr{P}$, then we are done. Otherwise, the ray enters a new geodesic triangle, which we handle by the same procedure.

As we showed earlier, at most $2 \log n$ kites, and hence geodesic triangles, can be crossed by the ray; therefore the query time is $O(\log^2 n)$. The only storage needed is for the balanced geodesic triangulation itself and its point-location structure, which is linear. The time to build the full data structure is $O(n \log n)$.

Note that the query time would go down to $O(\log n)$ if the geodesic triangles were real triangles instead. Of course, we can triangulate each geodesic triangle, but the crossing number (i.e., the number of edges crossed by a line segment) might increase in the process. We can limit the increase to a multiplicative factor of $\log n$ by a careful choice of triangulations. This will result in a triangulation of $\mathscr{P}$ such that any line segment interior to $\mathscr{P}$ intersects only $O(\log^2 n)$ edges. This triangulation may contain Steiner points, but it is nevertheless a proper cell complex and therefore can be navigated with $O(1)$ cost to go from one triangle to an adjacent one. This provides an alternative to the ray-shooting algorithm, which bypasses the binary searches needed to intersect concave chains with lines. This is explained in Section 4.

3. Building a Geodesic Triangulation in $O(n)$ Time. In this section we show how to build a balanced geodesic triangulation in optimal $O(n)$ time. The construction uses more sophisticated tools than the $O(n \log n)$ construction, and would consequently be harder to implement. Nevertheless, it is simpler than the preprocessing phase of the algorithm in [7].

Our construction is based on the shortest-path data structure of Guibas and Hershberger [11], [15]. That data structure can be built in $O(n)$ time (assuming linear-time triangulation of $\mathscr{P}$ [3]) and supports logarithmic-time shortest-path queries. A shortest-path query specifies two points inside the simple polygon and asks for the length of the shortest path between the points. As part of computing the path length, the query algorithm computes an implicit representation of the path; if the edges of the path are needed, they can be extracted in $O(\log n + k)$ time, where k is the number of edges. We build a geodesic triangulation by locating the *corners* of each geodesic triangle (the junctions between the concave chains), then perform shortest-path queries to link up the corners.

The time bound for shortest-path queries can be improved in the special case of computing a geodesic triangulation. By a mechanism that need not concern us here, the shortest-path data structure assigns a *height* to each diagonal in the triangulation of $\mathscr{P}$. The maximum height is $O(\log n)$; the total number of diagonals with height h is at most $O(n(\frac{2}{3})^h)$ [11]. If the triangles containing the two query points are known (as they are in our special case), the algorithm can compute the shortest path in time $O(h^2)$, where h is the height of the highest diagonal that intersects the shortest path. (If $h^2 > \log n$, this can be improved to $O(\log n)$.) The edges of the path can be extracted in $O(h^2 + k)$ time, where k is the number of edges [11], [15].

The same improvement applies to *funnel queries*. A funnel query specifies a point p and a triangulation diagonal; it asks for an implicit representation of the *funnel* they define [2], [12], [17]. This funnel is simply the geodesic triangle defined by p and the diagonal endpoints. The corner closest to p is the funnel *apex*. Funnels are used internally in the shortest-path data structure, so they are just as easy to compute as shortest paths. The two concave chains of the funnel are represented by binary trees of height $O(h)$, so we can compute tangents to them in $O(h)$ time [13], [20].

We use funnel queries to help compute geodesic triangles. To determine which funnels to construct, consider the dual tree of the triangulation, call it $\mathscr{D}$. For each of the three endpoints of a kite, we pick a representative triangle to which it is incident. These three triangles map to three nodes in $\mathscr{D}$. Exactly one node of $\mathscr{D}$ is incident to all three paths joining these nodes. Call this node and its corresponding triangle the *junction triangle*, and denote it by τ. The junction triangle is not necessarily unique—it depends on the choice of representative triangles—but it nevertheless helps us find the geodesic triangle corners. The following lemma uses the concept of a *sleeve*, which is a sequence of triangles corresponding to a path in $\mathscr{D}$.

LEMMA 3.1. *Let x, y, and z be the endpoints of a kite, let τ_x, τ_y, and τ_z be their representative triangles, and let τ be the junction triangle. Let $\mathscr{S}$ be the sleeve of*

triangles from τ_x through τ. Consider the two shortest paths from x to y and to z. The last common vertex of the two paths (one corner of the geodesic triangle) is contained in $\mathcal{S}$.

PROOF. Let $\mathcal{S}_{xy}$ be the sleeve of triangles dual to the path from τ_x to τ_y; define $\mathcal{S}_{xz}$ analogously. The shortest paths from x to y and from x to z are contained in the sleeves $\mathcal{S}_{xy}$ and $\mathcal{S}_{xz}$, respectively [17]. Therefore, the intersection of the two shortest paths, which perforce contains their last common vertex, is contained in $\mathcal{S}_{xy} \cap \mathcal{S}_{xz}$. However, $\mathcal{S} = \mathcal{S}_{xy} \cap \mathcal{S}_{xz}$. □

The following lemma lets us compute junction triangles quickly.

LEMMA 3.2. *After linear-time preprocessing, we can find the junction triangle of any three kite endpoints in constant time.*

PROOF. We preprocess the dual tree $\mathcal{D}$ for lowest common ancestor queries [14], [22]. Let τ_x, τ_y, and τ_z be the nodes of $\mathcal{D}$ representing the kite endpoints. We compute all three pairwise lowest common ancestors to get a multiset

$$A = \{\text{lca}(\tau_x, \tau_y), \text{lca}(\tau_x, \tau_z), \text{lca}(\tau_y, \tau_z)\}.$$

An easy case analysis shows that exactly one element of A occurs with odd multiplicity, and that element is the junction triangle τ. □

Once we know the junction triangle for three points, we can find their geodesic triangle using funnel queries. For each kite endpoint outside the junction triangle, we find the funnel from the point to the nearest edge of the junction triangle. By Lemma 3.1, the geodesic triangle corner closest to each endpoint lies in its funnel or in the junction triangle. To find the corner we compute the inner common tangents between pairs of funnels and between funnels and any kite endpoints inside the junction triangle. The inner common tangents determine the shortest paths between the kite endpoints, and hence determine the geodesic triangle. The two tangents to a funnel bound a chain of funnel vertices; the chain vertex closest to the funnel apex is a geodesic triangle corner (unless the geodesic triangle is empty, which can be determined from the inner common tangents). Once we know the corners of the geodesic triangle, we compute the shortest paths between them to produce the geodesic triangle edges. The complete construction cost for a single geodesic triangle is $O(h^2 + k)$, where h is the height of the highest diagonal the kite intersects (we say the kite has height h) and k is the number of edges bounding the geodesic triangle.

LEMMA 3.3. *In a balanced geodesic triangulation the total number of kites with height h is $O(nh(2/3)^h)$.*

PROOF. The total number of triangulation diagonals with height h is $O(n(\frac{2}{3})^h)$ [11]. The endpoints of the paths added at one stage of the geodesic triangulation process occur in sequence around the boundary of $\mathcal{P}$, and therefore the paths added at one stage intersect each diagonal at most twice. It follows that at most $O(\min(n(\frac{2}{3})^h, 2^j))$ kites of height h are added at stage j. Summing this over $1 \le j \le \log n$, we get

$$O\left(\sum_{j=1}^{\log n - h\log(3/2)} 2^j + \sum_{j=\log n - h\log(3/2)}^{\log n} n(\tfrac{2}{3})^h\right) = O(nh(\tfrac{2}{3})^h). \qquad \square$$

This lemma leads directly to the main theorem of this section.

THEOREM 3.4. *A balanced geodesic triangulation can be built in $O(n)$ time.*

PROOF. The cost of computing the geodesic triangulation is $O(n)$ for preprocessing, plus the cost of computing all the geodesic triangles. The cost of one geodesic triangle is $O(h^2 + k)$, where h is the height of the kite and k is the complexity of the geodesic triangle. The sum over all geodesic triangles of the second term is $O(n)$. By Lemma 3.3, the sum of the first term is

$$O\left(\sum_{h \ge 1} h^2 \cdot nh(\tfrac{2}{3})^h\right) = O(n). \qquad \square$$

4. A Real Triangulation with Steiner Points. Before we describe how to triangulate general geodesic triangles, let use consider the case of a *boomerang*, which is a polygon consisting of a V-shaped two-edge path connected by a concave polygonal path (Figure 4). Let m denote the number of sides of this (special) geodesic triangle. We choose a middle edge on the concave chain and extend it

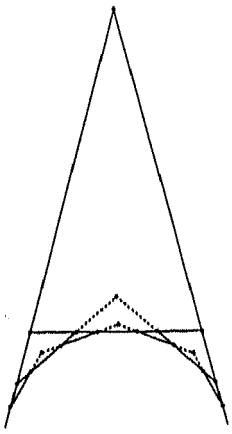

Fig. 4. Triangulating a boomerang.

in both directions until it meets the V-shaped path. We iterate on this process with respect to the two boomerangs created by this added line; the only difference lies in the fact that we extend the lines past the V-shaped paths (through the chord previously drawn) until they meet together. We repeat this operation with respect to the four new boomerangs arising from this construction and iterate the procedure in this fashion until the boomerangs have concave chains made of only one or two edges.

A more intuitive way to look at the construction is to regard each level k in the recursion as adding a concave polygonal chain of size at most 2^{k-1}. These chains do not share any edges but they intersect heavily (any two of them do, actually). It is easy to see, however, that the faces of the resulting map have at most six edges. The reason is that a face is either a (terminal) boomerang of constant size or the intersection of a boomerang with the complement of another boomerang (at one level lower) and a half-plane. We can therefore complete the triangulation of a boomerang trivially by refining each face.

What is the cost of computing the triangulation? From our last observation concerning the bounded size of the faces it suffices to analyze the time needed to add in the concave chains. For the same reason, intersecting the kth chain with all the previous ones can be done by traversing the chain in question in the current map, which will take time proportional to the number of intersections between the chain and all the ones previously drawn. Since the intersection of two concave chains consists of a number of points no greater than twice the size of the smaller chain, the cost of inserting the kth chain is $O(2^k)$; therefore computing the entire triangulation of the boomerang takes linear time. Because each of the concave chains can be intersected by a line segment at most twice and the number of such chains is at most $\log m$, the number of triangles that can be crossed by any segment is also $O(\log m)$.

We are now ready to triangulate a geodesic triangle Δ (Figure 5). To do so, we extend and connect in sequence the edges adjacent to the three "convex" vertices u, v, w of Δ, which creates three Steiner points, a, b, c. Next we draw the triangle abc and triangulate the three boomerangs created in the manner we just described. We now almost have a valid triangulation. We say "almost" because the three triangles adjacent to abc may have two of their sides broken up into $O(\log m)$ collinear edges by the way the boomerangs were triangulated. In this event, we

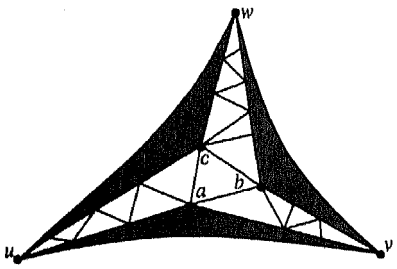

Fig. 5. Triangulating a geodesic triangle.

complete the triangulation by connecting the breakpoints together in a naive way, which adds only $O(\log m)$ edges to the current map. This also covers the case in which one chain of the geodesic triangle consists of only a single edge. The resulting triangulation of Δ is such that any segment in its interior can intersect only $O(\log m)$ edges. We summarize our findings below.

THEOREM 4.1. *Given a simple polygon of n vertices, it is possible to triangulate it into $O(n)$ triangles, using Steiner points, in $O(n)$ time, so that any interior line segment intersects only $O(\log^2 n)$ triangles. A much simpler algorithm runs in $O(n \log n)$ time.*

REMARKS. (1) Note that each Steiner point in the triangulation is determined by four vertices of the original polygon, that is, it is the intersection of two lines spanned by two vertices each. This implies that there are no hidden costs in the accurate representation of the triangulation. One should also note that Steiner points are sometimes necessary because otherwise a triangulation of a boomerang has a linear crossing number.

(2) We note that, for any triangulation (with or without Steiner points) of a convex n-gon, there is an interior line segment that intersects $\Omega(\log n)$ triangles. This follows from a result in [4]. It would be interesting to close the gap that remains for arbitrary simple polygons.

5. An Optimal Ray-Shooting Algorithm. The shortcoming of our previous solution is the need for repeated binary searches arising in the traversal of geodesic triangles, or, alternatively, the additional cost of navigating through the triangles of the real triangulation described in the previous section. We review that phase of the algorithm carefully and split it into two parts, which can be handled using fractional cascading and weight-balanced trees, respectively.

Typically, the query ray attempts to traverse a geodesic triangle Δ right after entering from the outside. Since we know the entering point, however, it suffices to consider the more general case in which we shoot from within Δ. The manner in which the ray relates to each of the three concave chains allows us to distinguish between a *fly-by* situation, where the ray is parallel to a tangent to a chain, and a *home-in* situation, where the slope of the ray falls outside the range of slopes specified by the end edges of the chain and the ray is directed toward the chain (Figure 6). In general, there is a fly-by situation for one of the chains and a home-in for another. There could also be no fly-by situation at all if the slope of the ray falls in the range between the two edges incident upon a "convex" vertex of Δ. We can easily check that the fly-by and home-in chains are the only ones through which the ray might leave Δ. Furthermore, it is easy to determine the type of a concave chain in constant time, given the query ray.

A concave chain C that presents the query ray with a home-in situation can be tested for intersection by binary search. Consider a balanced binary tree whose nodes are associated with the vertices of C. Because of the relative positioning of the ray and the chain, we know that they intersect in a single point. We can find

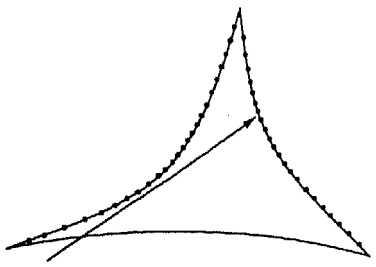

Fig. 6. The ray flies by the left upper chain and homes in toward the right upper chain.

this point in logarithmic time by following a single path down the tree, checking at each node on which side of the corresponding vertex the line is passing. The problem with using a balanced binary tree is that the search cost is the same regardless of whether the ray leaves Δ to enter a big or a small bay. (We use the word *bay* to refer to the portion of $\mathcal{P}$ separated from Δ by one of the edges of a chain; its *size* is the number of its edges.) Instead we use a weight-balanced binary search tree [19], in which each leaf is weighted by the size of the bay attached to the corresponding edge of the chain. If W is the total weight of all the bays attached to C and w is the weight of a particular bay, the cost of discovering that bay (i.e., its separating edge) is $O(1 + \log(W/w))$.

Let us now turn to the fly-by scenario. Instead of setting up a separate data structure for each chain, we combine all the needed information into a single array. Let a be the (usually unique) vertex of Δ that admits a vertical tangent. A line rolling around the three concave chains of the geodesic triangle covers all possible slopes from $-\infty$ to $+\infty$ in sequence, so if we store the edges of Δ in an array starting at a, we are in a position to find the tangential point of the fly-by chain in logarithmic time by binary search with the slope of the ray. Having found that point, we must decide whether the fly-by is successful or instead crashes into the concave chain, and if the latter, find the crashing point. To find the crashing point we use the home-in weight-balanced search tree for the chain. The difficulty caused by the fact that in general testing the query ray against a vertex of the chain no longer allows us to decide where to branch next is easily circumvented: indeed, the fly-by search has already identified a portion of the chain in which the hit must take place and the ray is now in a home-in situation with respect to that portion. (That portion lies between one of the endpoints of the chain and the vertex with a tangent parallel to the query ray.) Therefore, a test against a vertex within that portion is handled as in a typical home-in search, whereas a test against any other vertex requires no work since we already know which way to branch.

We now have (almost) all the ingredients to speed up the ray-shooting process. Let us momentarily forget about the fly-by searches and assume that they come for free. Let Δ be the triangle containing the starting point of the ray. Beginning at Δ, the query ray traverses geodesic triangles associated with a path of $k \leq 2 \log n$ nodes in the decomposition tree. Let $n_1, n_2, \ldots, n_k$ be the sizes of the successive

bays into which the query ray enters. Thus the cost of the ray shooting is at most proportional to

$$\log n + \sum_i \log \frac{n_i}{n_{i+1}},$$

which is $O(\log n)$.

Note that if we reverse the direction in which the ray is traced, from the hitting point back to the starting point, the total cost of traversing the weight-balanced trees is also $O(\log n)$. In our original traversal when we cross into a geodesic triangle through an edge e, we can therefore afford to walk up the path from e to the root of the weight-balanced tree associated with the chain containing e, before homing into the exit chain, because the total length of all these paths is still $O(\log n)$. This observation is used to speed up the fly-by operations, which we consider next.

To speed up the fly-by operations we modify the data structure one last time. The idea is to link all the fly-by arrays together and apply fractional cascading. Specifically, take the three weight-balanced trees associated with each geodesic triangle and make them the children of a fictitious root. Now connect each leaf of this new tree, representing some edge e, to the leaf representing e in the tree associated with the geodesic triangle (if any) right across e (Figure 7). It is immediate that this links all the weight-balanced trees into one big free tree, each of whose nodes has degree at most three. Finally, assign each fly-by array (a "catalog" in fractional cascading parlance) to the fictitious root corresponding to its geodesic triangle and assign an empty catalog to every other node in the free tree. In linear time we can apply fractional cascading to this catalog graph and reduce the cost of searching k catalogs along a path of the free tree from $O(k \log n)$ to $O(k + \log n)$ [5], [6]. By the observation made in the preceding paragraph, $k = O(\log n)$ in a ray-shooting query. Hence the overall time it takes to answer a query is $O(\log n)$.

We have thus achieved our main goal, which we summarize below.

THEOREM 5.1. *Given a simple polygon of n vertices, it is possible to preprocess it into a data structure of linear size, in $O(n)$ time, so that ray shooting within the polygon can be performed in $O(\log n)$ time. A more practical algorithm takes $O(n \log n)$ preprocessing time.*

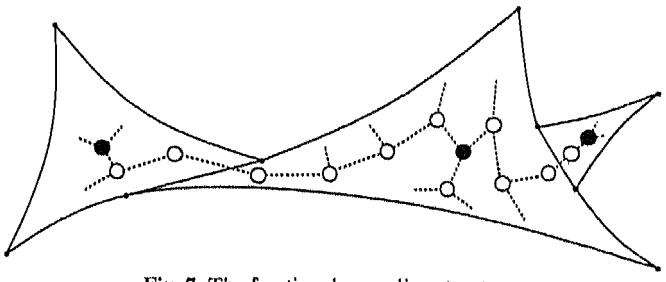

Fig. 7. The fractional cascading structure.

Note that despite the added complication of applying fractional cascading and weight-balanced trees, this solution is considerably simpler than the optimal method given in [7].

The method can be used to compute the visibility region from a point in a simple polygon in an output-sensitive manner. Given a point p inside a simple n-gon P, let V be the star-shaped polygon formed by the points in P that are visible from p and let k be the size of V. It is an elementary exercise to apply our ray-shooting algorithm to compute V in time $O(k \log n)$. The idea is to discover the geodesic triangles visible from p one at a time, in a graph-traversal fashion, by shooting rays from p. The bound follows from the fact that only $O(k)$ ray-shooting queries are necessary. Note that this solution outperforms the $O(n)$-time algorithm of ElGindy and Avis [10] unless k is close to linear.

6. Ray Shooting Amidst Polygonal Obstacles. Assume now that we are given k disjoint simple polygons with a total of n vertices and that we want to perform ray shooting from any point *outside* these polygons. We show how to do this in time $O(\sqrt{k} \log n)$ and storage $O(n)$, using $O(n\sqrt{k} + k^{3/2} \log k + n \log n)$ preprocessing. To begin with, we choose one vertex in each polygon and we connect them into a spanning tree $\mathcal{T}$ consisting of nonintersecting line segments. By the methods of Matoušek [18], we can compute such a tree in $O(k^{3/2} \log k)$ time, while at the same time ensuring that any line can intersect only $O(\sqrt{k})$ edges of $\mathcal{T}$. Although the edges of $\mathcal{T}$ are mutually disjoint (except at their endpoints) they might cross the polygons. This would be very undesirable for what we have in mind, which is to connect the k polygons into a single simple polygon. So we must see how the edges of $\mathcal{T}$ intersect the polygons and remove pieces of them to make sure that no such crossings remain.

To do this, we first triangulate each of our polygons, and also their common exterior, in $O(n \log n)$ time, into a total of $O(n)$ triangles. Then we take each edge of $\mathcal{T}$ and traverse it through the triangulation, to obtain all its intersections with the edges of the triangulation. Since each triangulation edge can intersect at most $O(\sqrt{k})$ edges of $\mathcal{T}$, there will be a total of at most $O(n\sqrt{k})$ intersection points, and they can all be found in time $O(n\sqrt{k})$.

The intersection points partition the edges of $\mathcal{T}$ into subsegments, each of which either lies inside one of our polygons, or lies outside the polygons but connects two points on the same polygon boundary, or connects the boundaries of two distinct polygons. Moreover, by sorting these intersection points along the edges of $\mathcal{T}$, which in essence is already done, it is easy to determine the type of each subsegment, including the polygons that contain its endpoints. We now consider only subsegments of the third type, and add them one by one, maintaining the connected components of the union of all polygons and subsegments already inserted. This is done using a simple union-find structure for the set of polygons, and allows us to discard subsegments that connect two polygons that already lie in the same component. When we are all done we enclose the resulting region (namely the union of all polygons and connecting subsegments) in a sufficiently

large rectangle, which we connect to the region by a line segment. Hence (the portion within the rectangle of) the complement of our region becomes a simple polygon $\mathscr{P}$, provided we regard the added subsegments as infinitesimally thin slabs. (This is, by the way, also the method we use above to convert $\mathscr{T}$ into a simple polygon that is amenable to preprocessing for ray shooting.) Finally, we preprocess the resulting $\mathscr{P}$ for fast ray shooting as in the preceding section. The total preprocessing time is clearly $O(n\sqrt{k} + k^{3/2} \log k + n \log n)$ and the storage needed is linear. Since a query ray can intersect only $O(\sqrt{k})$ edges of $\mathscr{T}$, and from each such intersection we shoot a new ray, the query time is $O(\sqrt{k} \log n)$.

As we did in Section 3 we can triangulate the geodesic triangles of the final decomposition in order to obtain a triangulation such that any line segment exterior to any polygonal obstacle intersects only $O(\sqrt{k} \log^2 n)$ edges.

THEOREM 6.1. *Given a collection of k disjoint simple polygons with a total of n vertices, it is possible to preprocess it, in $O(n\sqrt{k} + k^{3/2} \log k + n \log n)$ time, into a data structure of size $O(n)$, so that ray shooting in the common exterior of the polygons can be performed in $O(\sqrt{k} \log n)$ time.*

REMARK. The query time of the algorithm is faster, by a factor of $O(\log n)$, than that of the previously best algorithm, due to Agarwal [1].

Acknowledgments. The authors would like to express their gratitude to the DEC Systems Research Center in Palo Alto, where most of the work on the paper was carried out, for its generous hospitality and support.

References

[1] P. Agarwal, Ray shooting and other applications of spanning trees with low stabbing number, *Proc. 5th ACM Symp. on Computational Geometry*, 1989, pp. 315–325.
[2] B. Chazelle, A theorem on polygon cutting with applications, *Proc. 23rd IEEE Symp. on Foundations of Computer Science*, 1982, pp. 339–349.
[3] B. Chazelle, Triangulating a simple polygon in linear time, *Discrete Comput. Geom.*, **6** (1991), 485–524.
[4] B. Chazelle, H. Edelsbrunner, and L. Guibas, The complexity of cutting complexes, *Discrete Comput. Geom.*, **4** (1989), 139–181.
[5] B. Chazelle and L. Guibas, Fractional cascading: I. A data structuring technique, *Algorithmica*, **1** (1986), 133–162.
[6] B. Chazelle and L. Guibas, Fractional cascading: II. Applications, *Algorithmica*, **1** (1986), 163–191.
[7] B. Chazelle and L. Guibas, Visibility and intersection problems in plane geometry, *Discrete Comput. Geom.*, **4** (1989), 551–581.
[8] D. Dobkin and D. Kirkpatrick, Fast detection of polyhedral intersection, *Theoret. Comput. Sci.*, **27** (1983), 241–253.
[9] H. Edelsbrunner, L. Guibas, and J. Stolfi, Optimal point location in a monotone subdivision, *SIAM J. Comput.*, **15** (1986), 317–340.

[10] H. ElGindy and D. Avis, A linear algorithm for computing the visibility polygon from a point, *J. Algorithms*, **2** (1981), 186–197.

[11] L. Guibas and J. Hershberger, Optimal shortest path queries in a simple polygon, *J. Comput. System Sci.*, **39** (1989), 126–152.

[12] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan, Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons, *Algorithmica*, **2** (1987), 209–233.

[13] L. Guibas, J. Hershberger, and J. Snoeyink, Compact interval trees: a data structure for convex hulls, *Internat. J. Comput. Geom. Appl.*, **1** (1991), 1–22.

[14] D. Harel and R. E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.*, **13** (1984), 338–355.

[15] J. Hershberger, A new data structure for shortest path queries in a simple polygon, *Inform. Process. Lett.*, **38** (1991), 231–235.

[16] D. Kirkpatrick, Optimal search in planar subdivisions, *SIAM J. Comput.*, **12** (1983), 28–35.

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

[18] J. Matoušek, More on cutting arrangements and spanning trees with low stabbing number, Technical Report B-90-2, Freie Universität Berlin, February 1990.

[19] K. Mehlhorn, *Data Structures and Algorithms, I: Sorting and Searching*, Springer-Verlag, Heidelberg, 1984.

[20] M. Overmars and J. van Leeuwen, Maintenance of configurations in the plane, *J. Comput. System Sci.*, **23** (1981), 166–204.

[21] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, Heidelberg, 1985.

[22] B. Schieber and U. Vishkin, On finding lowest common ancestors: simplification and parallelization, *Proc. Third Aegean Workshop on Computing*, pp. 111–123, Lecture Notes in Computer Science, Vol. 319, Springer-Verlag, Berlin, 1988.