Discrete Comput Geom 27:461–483 (2002)
DOI: 10.1007/s00454-001-0089-9

Discrete & Computational
Geometry

Visibility Queries and Maintenance in Simple Polygons*

B. Aronov,[1] L. J. Guibas,[2] M. Teichmann,[3] and L. Zhang[2†]

[1]Department of Computer and Information Science, Polytechnic University,
Brooklyn, NY 11201-3840, USA
aronov@ziggy.poly.edu

[2]Computer Science Department, Stanford University,
Stanford, CA 94305, USA
guibas@cs.stanford.edu

[3]Lateral Logic Inc.,
Montreal, Quebec, Canada.
marekt@mit.edu

Abstract. In this paper we explore some novel aspects of visibility for stationary and moving points inside a simple polygon P. We provide a mechanism for expressing the visibility polygon from a point as the disjoint union of logarithmically many canonical pieces using a quadratic-space data structure. This allows us to report visibility polygons in time proportional to their size, but without the cubic space overhead of earlier methods. The same canonical decomposition can be used to determine visibility within a frustum, or to compute various attributes of the visibility polygon efficiently. By exploring the connection between visibility polygons and shortest-path trees, we obtain a kinetic algorithm that can track the visibility polygon as the viewpoint moves along polygonal paths inside P, at a polylogarithmic cost per combinatorial change in the visibility or in the flight plan of the point. The combination of the static and kinetic algorithms leads to a new static algorithm in which we can trade off space for increased overhead in the query time. As another application, we obtain an algorithm which computes the weak visibility polygon from a query segment inside P in output-sensitive time.

* A preliminary extended abstract appeared in *Proc. 9th Annual International Symposium on Algorithms and Computation*, pp. 327–336, 1998. B. Aronov was partially supported by NSF Grant CCR-92-11541 and a Sloan Research Fellowship. L. J. Guibas and L. Zhang were partially supported by NSF Grants CCR-9623851, CCR-9910633, U.S. Army Research Office MURI Grant DAAH04-96-1-0007, and DARPA TMR Grant DAAE07-98-L027. M. Teichmann was supported by the National Science and Engineering Council of Canada.

† Current address: Compaq Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA. l.zhang@compaq.com.

1. Introduction

In this paper we consider two visibility-related problems in simple polygons—one is the visibility-query problem from a fixed observer; the other is the problem of maintaining the visibility from a moving observer. In the former problem, we are given a simple polygon P, and we want to process it into a data structure so that, for any query point q inside P, the visibility polygon $V(q)$ from q can be reported efficiently. In the latter problem we would like to maintain this visibility polygon while the viewpoint moves along linear segments. In the following discussion the size of a polygon is understood as the number of its vertices. We denote by n the size of P and by $|V(q)|$ the size of $V(q)$.

There have been many studies on computing visibility polygons in a simple polygon. An algorithm with linear running time is first achieved in [7]. Some other linear algorithms based on triangulation are given in [5] and [10]. In the worst case these algorithms are optimal as visibility polygons may have up to linear complexity. However, in practice, the visibility polygons are usually much less complex than the environment, and, in many applications, we would like to compute the visibilities from many different viewpoints. In these cases it is desired to make the running time of the algorithm output-sensitive. That is, we would like the running time of our algorithm to be proportional to the size of the output, after certain preprocessing of the environment.

In [17] an algorithm with $O(|V(q)|\log(n/|V(q)|))$ query time after a preprocessing stage of $O(n^2)$ space and time is given. Another method to achieve output sensitivity is by building a linear-size, $O(\log n)$-time ray-shooting query structure as in [12] and shooting rays to discover the visibility polygon. This way, one can construct an algorithm that requires only linear storage and preprocessing time and $O(|V(q)|\log n)$ query time for a query point q. Note the appearance of a multiplicative overhead in the query time bounds of the above algorithms. In [2] and [11] an optimal query time without multiplicative overhead is achieved. The query time of their algorithms is $O(\log n + |V(q)|)$, but the storage and preprocessing time is cubic. If the queries are restricted to a given line segment, then there exists an algorithm that reports $V(q)$ in $O(\log n + |V(q)|)$ time for any query point q after $O(n\log n)$ time preprocessing [6]. However, the method in [6] does not extend to the general case.

A general technique used to answer visibility queries is to decompose the interior of a polygon into regions so that points in the same region see "equivalent" visibility polygons. A point location structure built on such a decomposition is then used to answer queries. In [2] and [11] the $O(\log n + |V(q)|)$ query time is achieved by storing such a decomposition in $O(n^3)$ space. In this paper we show how to decompose the polygon into canonical pieces in order to keep their visibility information separately, so as to reduce the storage and preprocessing time. As a result, our algorithm constructs a data structure of size $O(n^2)$ which can be computed in time $O(n^2\log n)$ so that the visibility polygon $V(q)$ from any query point $q \in P$ can be reported in $O(\log^2 n + |V(q)|)$ time. Note that in our algorithm, there is no multiplicative overhead in the query time. In addition, our method for finding the visibility polygon expresses this polygon as the union of $O(\log n)$ canonical pieces, which need not be constructed explicitly unless needed. By exploiting this fact we are able to answer efficiently additional types of queries. For example, we can report the (combinatorial) size of the visibility polygon in $O(\log^2 n)$ time, and have

output-sensitive visibility queries when visibility is delimited by a cone centered at the observer and defining the viewing frustum.

We also consider the problem of maintaining visibility from a linearly moving viewpoint. This problem appears in many typical application settings, such as architectural walk-throughs, where one wants to compute the visibility from a continuously moving viewpoint. A similar problem is studied in [6]. In that paper the point moves along a given line segment, so that the direction of the motion is fixed. Here, we allow the motion of the point be updated in an on-line fashion. By exploiting the intimate connection between visibility and shortest-path trees, we obtain an algorithm that can maintain the visibility polygon from a point which moves along linear segments, using linear space and $O(\log^2 n)$ time per event. An event is defined to be either a (combinatorial) change of the visibility polygon, or an update of the linear motion.

Lastly, we show some applications of the combination of our methods for visibility queries and maintenance of a visibility polygon with a moving viewpoint. In [2] it is asked if there exists a smooth tradeoff between preprocessing storage and query time for the visibility query problem. A combination of the above two procedures provides an algorithm with the tradeoff of $O(m)$ space, $O(m \log n)$ preprocessing time, and $O((n^2/m)\log^3 n + |V(q)|)$ query time for any m between $n \log^3 n$ and $n^2 \log n$. Our second application is to solving the weak visibility query problem. We provide a method for reporting the weak visibility polygon of a query line segment $s \subset P$ in output-sensitive time by combining both of our methods.

The paper is organized as follows. We first give some definitions and notation in Section 2. In Section 3 we show how to answer visibility queries in nearly optimal time. The algorithm which maintains the visibility polygon from a moving viewpoint is given in Section 4. In Section 5 we describe the applications.

2. Preliminaries

2.1. *Definitions and Geometric Facts*

Suppose P is a simple polygon. Denote the boundary of P by ∂P. ∂P consists of vertices and edges. Throughout this paper we assume polygons are in general position, i.e., no three vertices of P are collinear. Degenerate cases can be handled by standard perturbation techniques.

Two points p, q can *see* each other if the line segment $\overline{pq}$ lies in the polygon. If two boundary points p, q are mutually visible, $\overline{pq}$ is called a *chord*. A chord is also called a *diagonal* if both its endpoints are vertices. A chord s separates a simple polygon into two connected components. If two points are in the same component, they are also called on the same side with respect to s. *The visibility polygon* $V(p)$ of a point $p \in P$ consists of all the points that can be seen from p. A visibility polygon is a star-shaped polygon whose boundary consists of parts of polygon boundaries and chords, called *windows*. Any point invisible to p is separated from $V(p)$ by a window.

A point p can *see* a subset S inside P if p sees at least one point in S. Similarly, define the *weak visibility polygon* of S, denoted by $V(S)$, to be the set of points in P which can see S, or, equivalently, $V(S) = \bigcup_{p \in S} V(p)$ (Fig. 1).

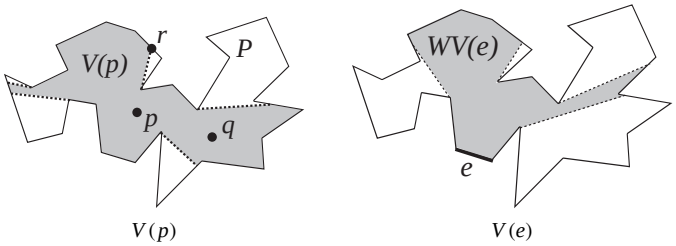


Fig. 1. Visibility polygon and weak visibility polygon. The shaded regions are $V(p)$ and $V(e)$, respectively. The dotted segments on the boundary of the visibility polygons are windows.

A visibility polygon can be represented geometrically by a circular list of the vertices with their planar coordinates. With the prior knowledge of the polygon P, one can represent visibility polygons combinatorially by a circular list of vertices and edges of P in the order in which they appear on the boundary of the visibility polygon. This list is called the *combinatorial representation* of $V(p)$. The actual coordinates of each vertex of $V(p)$ can be computed in $O(1)$ time given the combinatorial representation. From now on, when we refer to a visibility polygon, we mean its combinatorial representation, unless otherwise stated. Two visibility polygons are (combinatorially) *equivalent* if their combinatorial representations are identical (up to a circular permutation). An important property of simple polygons is that once the vertices and edges visible from a point are known, the order in which they appear on the visibility polygon is uniquely determined, as it coincides with the order along the boundary of the original polygon. This property implies that if two points see the same set of vertices and edges of P, then they have equivalent visibility polygons. Note that this statement breaks down for non-simple polygons.

In a simple polygon the portion of a line segment s visible to a point p or weakly visible to another segment s' is, if non-empty, a contiguous subsegment of s. We denote by $C_s(p)$ the infinite cone with apex p and delimited by the endpoints of the visible portion of s.

A line ℓ is *tangent* to a polygon P at vertex v if ℓ passes through v, and ℓ is inside P in an open neighborhood of v. Notice that a line can be tangent to P only at a reflex vertex. For a point $p \in P$ and a vertex v, consider the ray emanating from p, aimed at v, and crossing ∂P at a point w after v ($w \neq v$). If $\overline{pw}$ is inside P and the line containing $\overline{pw}$ is tangent to P at v, the chord $\overline{vw}$ is called the *constraint* induced by p and v (we say it is "induced by p" when the vertex v is unimportant or understood from context) and is denoted by $c(p, v)$. An easy observation is that for any point p inside P, the constraints induced by p are exactly the windows of $V(p)$. Further, when both p and v are polygon vertices, the constraint $c(p, v)$ is called a *critical constraint* (Fig. 2). All the critical constraints partition the interior of the polygon into cells, which is called the *visibility decomposition* and denoted by $\mathcal{V}(P)$ (Fig. 2). According to the following lemma, the visibility decomposition decomposes the interior of P into cells with equivalent visibility polygons.

Lemma 2.1. *Suppose p, q are two points inside a simple polygon P and not on any critical constraint. They see different sets of vertices and edges if and only if they are on opposite sides of some critical constraint.*

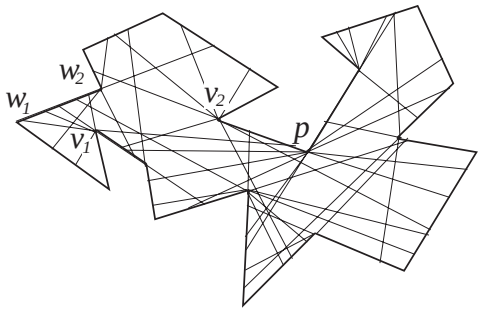


Fig. 2. The visibility decomposition of the polygon P; $\overline{v_1w_1}$, $\overline{v_2w_2}$ are the critical constraints reduced by p and v_1, v_2, respectively.

Proof. Consider all the visibility polygons of vertices of P and the weak visibility polygons of its edges. They are simple polygons whose boundaries consist of polygon edges and critical constraints. Thus, if p, q are on the same side for all the critical constraints, their visibility (weak visibility) from all the vertices (resp. edges) is the same. That is, they see the same set of vertices and edges.

On the other hand, if p, q see equivalent visibility polygons, we shall prove that no critical constraint separates them. We prove this by contradiction. Suppose that there is a critical constraint separating them. Let Δ_p, Δ_q denote the cells in the visibility complex that contain p and q, respectively. Then Δ_p and Δ_q must be different. Since they are both convex cells, there must be a critical constraint on the boundary of Δ_p separating them. Suppose $c(u, v)$ is on the boundary of Δ_p and separates Δ_p and Δ_q (Fig. 3). Clearly, v is visible from p. By the assumption that p, q see the same set of vertices, v is also visible from q. If the line uv separates the edges incident to v from Δ_p, then the edge e_1 incident to u is visible from p but not from q. If the edges incident to v are on the same side as Δ_p, then the extension of the line segment $\overline{qv}$ must hit a polygon edge, say e_2. Then e_2 is visible from q but not from p. In both cases we have derived contradiction. Therefore, there cannot be any critical constraint separating p and q if they see the equivalent visibility polygon. □

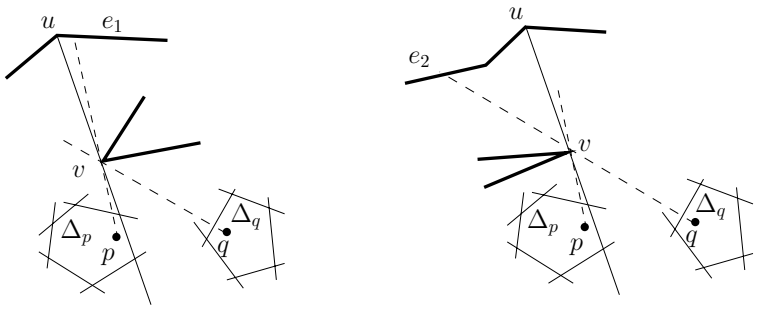


Fig. 3. Proof of Lemma 2.1.

Remark 2.1. The proof of the above "folklore" result is presented here because we could not find a proof for the "if" part in the literature, while there have been several proofs for the "only if" part.

Remark 2.2. In the above lemma we did not include the case when a point is on a critical constraint. However, a point on a segment has the visibility polygon equivalent to that of a point to one side of the segment—to which side to perturb depends on how the vertex on the critical constraint blocks the visibility.

Remark 2.3. Notice that in our definition the combinatorial representation consists of both the vertices and edges visible to a point. If only the vertices are considered, the above lemma is not true. It is very easy to construct an example where two points in different faces of the arrangement of critical constraints can see the same set of vertices.

Since there are $O(n^2)$ critical constraints, an immediate upper bound on the complexity of visibility decomposition is $O(n^4)$. However, the following fact implies a better bound of $O(n^3)$, which is tight in the worst case.

Lemma 2.2. *Any segment s inside a simple polygon P can cross at most $O(n)$ critical constraints of P.*

Proof. Refer to [2] and [11]. □

By Lemma 2.2, the number of vertices and, therefore, the complexity of the visibility decomposition is $O(n^3)$.

Another geometric object closely connected to visibility is the shortest path. The *shortest path* $\pi(p, q)$ between two points $p, q \in P$ is the path with the shortest Euclidean length among all the paths joining p, q inside P. The path $\pi(p, q)$ is a polygonal path in which all the intermediate vertices are reflex polygon vertices. The union of the shortest paths connecting p and all the vertices of P form a tree rooted at p. This rooted tree is known as the *shortest-path tree* (Fig. 4) and is denoted by $T(p)$. Clearly, if a vertex u is a child of p in the tree $T(p)$, then u is visible from p. From the shortest path tree $T(p)$, it is very easy to obtain the visibility polygon $V(p)$ [10].

In Section 4 we show how to maintain the shortest-path tree, and thus the visibility polygon, of a moving point.

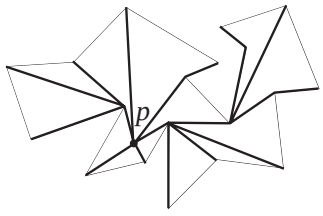


Fig. 4. The shortest-path tree $T(p)$ of a point p.

2.2. *Persistent Red-Black Tree*

In our method we use *persistent data structures* to reduce space cost. Further, in our tradeoff result, we need to extend the notion of a persistent data structure to handle *transient* dynamic updates. These data structures are described in this section.

The term *persistent data structure* was coined by Sarnak and Tarjan in [16]. In general, a persistent data structure is one that accepts an arbitrarily long sequence of updates, but is able to remember at any time all its earlier versions. Persistent data structures have proven very useful for storing a sequence of data sets with only slight changes between any adjacent two in the sequence. Here, we focus our attention on persistent red-black trees.

Suppose we have a set of n linearly ordered items and a sequence of m updates (i.e., insertions and deletions) of these items. Let the *version at time* t, for $1 \leq t \leq m$, be the set resulting from applying the first t updates in the sequence to an empty set. According to [16], a persistent red-black tree can be built so that any version can be accessed with the same time bounds as stored in a standard (ephemeral) red-black tree. Furthermore, the structure can be constructed in $O((m+n)\log n)$ time by using $O(m+n)$ space. In this paper, in addition to accessing a version, we are also interested in accessing the set obtained by applying some updates to any version. Formally, a query is a tuple of the form $(t, update_1, \ldots, update_k, acc)$. For such a query, we need to return the result of the access operation *acc* to the set that results from applying the sequence of updates $(update_1, \ldots, update_k)$ to the set of version t. The updates in such a query are not persistent, and we do not keep them after the desired accesses are made. We call such updates *transient*. To be able to perform transient updates on a persistent red-black tree, we proceed as follows. In each node we add fields which are labeled `transient` to hold the necessary data for an ephemeral red-black tree, i.e., the color and pointers pointing to the parent and children. We also add a one-bit field per node to indicate if the transient fields are used or not. During an update, whenever we need to insert a new item, we create a node labeled `transient` ; when we need to redirect a pointer or recolor a node, we store all the information in the transient fields without modifying the original data structure. We also keep track of all the places where an update has happened by linking all such nodes in a list. When we need to follow a pointer, we first check the indicator to see if the transient field has been updated. If it has, we follow the pointer stored in that transient field. Otherwise, we follow the one in the original data structure. After accesses are made, the data structures are cleaned up by deleting all the "transient" nodes and resetting all the `transient` indicators. The operations can be done in $O(\log n)$ time per transient update (and for the final access) because they can be viewed as ephemeral red-black tree operations.

To summarize, we have

Lemma 2.3. *A sequence of m updates of n linearly ordered items can be processed in time $O((m+n)\log n)$ into a data structure using $O(m+n)$ space so that each version can be accessed in the same time bound as a red-black tree. Further, transient updates can be made to any version in $O(\log n)$ time per up date.*

3. Answering Visibility Queries

In this section we provide an algorithm with $O(\log^2 n + |V(q)|)$ query time, which is nearly optimal and requires $O(n^2 \log n)$ preprocessing time and $O(n^2)$ storage. Compared with the algorithms in [2] and [11], the storage and preprocessing time has one fewer linear factor. Further, we show that the algorithm can be extended to handle the problems of counting the size of a visibility polygon and answering *cone-visibility* queries.

Intuitively, our algorithm works as follows: we first observe that when a point q is not inside a subpolygon P' of P, it is "easy" to compute the *partial visibility polygon* $V(q) \cap P'$. Then we compute a hierarchical representation of the polygon by using balanced triangulation hierarchies. For any query point q, we can decompose the polygon into $O(\log n)$ disjoint subpolygons, each represented by a node in the balanced triangulation tree, such that all the subpolygons, except for the triangle containing q, do not contain q. For each subpolygon P', we then compute the corresponding partial visibility polygon and glue all the partial visibility polygons together to obtain $V(q)$.

In Section 3.1 we show how to construct a data structure to answer a partial visibility polygon query efficiently. In Section 3.2 we describe the balanced triangulation and present the full algorithm and its extensions.

3.1. *Computing Partial Visibility Polygons*

For a polygon Q contained in P, define the *partial visibility polygon* $V_Q(q)$ to be the polygon $V(q) \cap Q$. Suppose P is divided into two parts, L and R, by a diagonal e. In this section we show how to compute the partial visibility polygon $V_L(q)$ for a point $q \in R$.

For an edge e, we denote by e^+ and e^- the half-space to the left and right side of the line passing e, respectively. We assume that L is to the left of ℓ, the line on which the diagonal e lies, that is, L is in e^+. In general, this is not true as L may "bend over" to cross ℓ. However, we can conceptually truncate L by the extension of e because q cannot see any part of L on the right side to ℓ—more precisely, L is the connected component of $P \cap e^+$ which contains e. Further, notice that if q lies in the same side as L to ℓ, then q cannot see any part of L, except for at most one vertex of e. Thus, in the following, we also assume that q lies to the right of ℓ, i.e., $q \in e^-$.

Observe that for any two points $p \in L$ and $q \in R$, q can see p if and only if $\overline{pq}$ does not cross ∂P. Since $\partial P = (\partial L \backslash e) \cup (\partial R \backslash e)$, we consider them separately as follows. Recall that the visibility cone $C_e(q)$ is delimited by the endpoints of the portion of e which is visible to q. Therefore, $\overline{pq}$ does not cross $\partial R \backslash e$ if and only if p lies inside the cone $C_e(q)$, because no portion of L can block the visibility of a point on e to $q \in R$. On the other hand, if we define the *exterior visibility polygon* $EV_L(q)$ of q with respect to L to be the portion of L which can be seen by q through e as if all the edges of ∂R are transparent, then $\overline{pq}$ does not cross $\partial L \backslash e$ if and only if $p \in EV_L(q)$. Therefore, the intersection between the point sets $C_e(q)$ and $EV_L(q)$ is exactly the set of all the points p where $\overline{pq}$ does not cross properly $\partial L \backslash e$ and $\partial R \backslash e$. That is, $V_L(q) = C_e(q) \cap EV_L(q)$. The above procedure is depicted in Fig. 5. In the following sections we show how to compute $C_e(q)$, $EV_L(q)$, and their intersection $V_L(q)$.

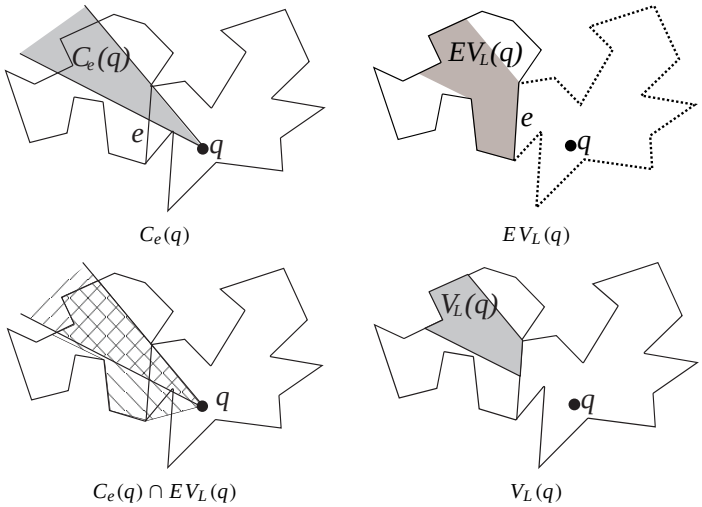


Fig. 5. Computing the partial visibility polygon.

3.1.1. *Computing* $C_e(q)$. Since P is a simple polygon, the portion of e visible to q is a line segment. To compute $C_e(q)$, we just need to find the endpoints of this segment or, in other words, the extremal points on e that q can see. By exploiting the connection between visibility and shortest-path trees, we can show that the extremal points can be computed in $O(\log n)$ time after linear time precomputation.

Suppose that v_1, v_2 are the endpoints of the edge e. Consider the shortest paths $\pi(q, v_1)$ and $\pi(q, v_2)$. Following the terminology of [10], $\pi(q, v_1)$, $\pi(q, v_2)$ form a *funnel* which may consist of a shared initial path $\gamma = \pi(q, v_1) \cap \pi(q, v_2)$ and a region bounded by two outward convex chains and the segment e. To compute the endpoints, we distinguish three cases (refer to Fig. 6):

1. There is no common initial path, i.e., $\gamma = \{q\}$. Let s_1, s_2 be the edges of $\pi(q, v_1)$, $\pi(q, v_2)$, respectively, incident to q. By the convexity of the funnel, the extensions of s_1, s_2 will meet the edge e without intersecting any other polygon edges. Thus, the intersection points delimit the portion visible to q.

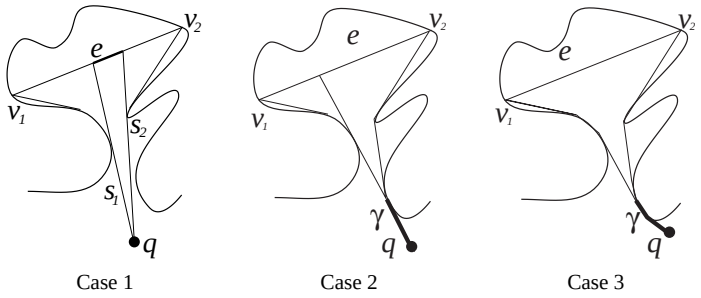


Fig. 6. Different cases in computing $C_e(q)$.

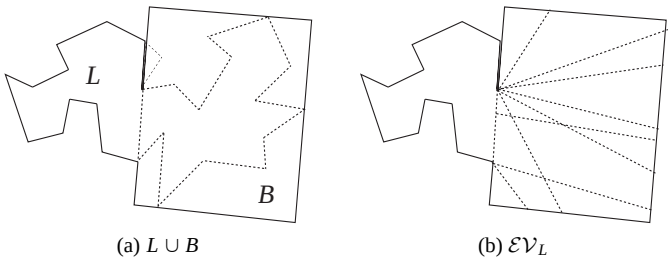
2. The path γ consists of a single segment, say s, and s is collinear with the adjacent segment on at least one of the two shortest paths. In this case, q sees exactly one point on e, namely, the intersection point between the extension of s and the edge e.
3. The path γ consists of a single segment and this segment is not collinear with any of its adjacent segments on the two shortest paths, or γ consists of several segments. In this case, q cannot see anything on e.

Thus, $C_e(q)$ can be computed by checking the first two edges on the shortest paths $\pi(q, v_1)$ and $\pi(q, v_2)$. In [9] it is shown that a data structure can be built in linear time by using linear space so that, for any two query points, the length of the shortest path can be reported in $O(\log n)$ time. Using the same structure with slight modification, the first two edges on the shortest path can be reported in $O(\log n)$ time as well. Thus, we have

Lemma 3.1. *Given a simple n-gon P, it can be processed into a structure in $O(n)$ space and $O(n)$ preprocessing time so that for any diagonal e of P and any query point, $C_e(q)$ can be computed in $O(\log n)$ time.*

As we will see later, the use of the shortest-path query data structure here is just for description convenience. We can actually obtain $C_e(q)$ as the algorithm proceeds, as remarked in Section 3.2.

3.1.2. *Computing $EV_L(q)$.* Recall that $L \subset e^+$ and $q \in e^-$. To compute $EV_L(q)$, similar to the visibility decomposition, we decompose e^- into cells so that two points in the same cell see equivalent visibility polygons in L. This decomposition is called *exterior visibility decomposition* and denoted by $\mathcal{EV}_L$. Once we have constructed $\mathcal{EV}_L$, $EV_L(q)$ can be computed by locating q in the cell decomposition and retrieving the corresponding exterior visibility polygon. The way that $\mathcal{EV}_L$ is formed is similar to that of the visibility decomposition. It can be regarded as a special case of the visibility decomposition. Imagine the bounding box B of R (i.e., a minimal rectangle containing R) with one side lying on ℓ. We form a simple polygon Q by taking the union of L and B (Fig. 7(a)). We then compute the visibility decomposition for this polygon and consider the decomposition clipped in the box B. Clearly, for any two points in B, if they are in the same cell of this decomposition, they will see the same visibility polygon of L. Further, it suffices to consider B only because $R \subset B$ (Fig. 7(b)).



(a) $L \cup B$ (b) $\mathcal{EV}_L$

Fig. 7. (a) The union of L and B. Note that L is clipped by the half-space e^+. (b) The visibility decomposition where only the part in B is drawn.

Which critical constraints can contribute to form the decomposition of B? The answer is that they must be those induced by a pair of vertices in L and crossing the diagonal edge e. According to Lemma 2.2, the number of such critical constraints is $O(n)$. Thus, effectively, we reduce the number of critical constraints under consideration from n^2 to n. Furthermore, those constraints can be computed in $O(n \log n)$ time as shown in [6]. By a topological sweep, their arrangement can be built in $O(n^2)$ time (Fig. 7(b)).

For each of the cells in $\mathcal{EV}_L$, we compute and store the corresponding visibility polygon. If implemented in a naive way, it may take $\Theta(n^3)$ space and preprocessing time because a visibility polygon may have $\Theta(n)$ complexity. However, note that any two adjacent cells have only $O(1)$ differences in their visibility polygons because they are separated by only one critical constraint. By using a persistent data structure, we can reduce the costs to $O(n^2 \log n)$ preprocessing time and $O(n^2)$ storage. More precisely, we form a dual graph $\mathcal{D}$ of the decomposition and compute a spanning tree of $\mathcal{D}$. By performing a depth-first traversal of the tree, we can obtain a tour visiting all the cells and traversing each edge of $\mathcal{EV}_L$ at most twice in $\mathcal{EV}_L$. Recall that each visibility polygon can be represented by a circular list of the visible vertices and edges in the order in which they appear on the boundary of the original polygon. $EV_L(q')$ can be obtained from $EV_L(q)$ by $O(1)$ updates if q, q' are points in adjacent cells.

Thus, we can start from an arbitrary node in the dual graph, walk along the tour, and construct a persistent red-black tree on the combinatorial representation of the visibility polygon for all the nodes. As per Lemma 2.3, the structure takes $O(n^2)$ storage and can be built in $O(n^2 \log n)$ preprocessing time. In addition, we also build a point location structure on top of the arrangement which can be done in $O(n^2)$ time and $O(n^2)$ space [13].

To answer a query q, we first locate the cell of $\mathcal{EV}_L$ in which q lies, and then retrieve the corresponding root pointer in the persistent data structure. Both steps can be done in $O(\log n)$ time. Once we obtain the root pointer, we can either report $EV_L(q)$ by traversing the tree or perform any other search in the tree. To summarize, we have

Lemma 3.2. *A simple n-gon L with a distinguished edge e, where $L \subset e^+$, can be processed into a data structure by using $O(n^2)$ space and $O(n^2 \log n)$ preprocessing time so that, for any query point $q \in e^-$, a pointer pointing to a red-black tree which stores $EV_L(q)$ can be returned in $O(\log n)$ time.*

3.1.3. *Computing $V_L(q)$.* Once we have computed $C_e(q)$ and (a pointer to a searchable representation of) $EV_L(q)$, $V_L(q)$ can be computed by extracting the portion of $EV_L(q)$ inside the cone $C_e(q)$. Because the visibility polygon is star-shaped, and the visible vertices and edges are stored in a red-black tree in the order of their appearance on the visibility polygon, the pruning procedure amounts to reporting all the elements of a red-black tree within a given range of keys. For the persistent data structure in [16], this can be done in $O(\log n + k)$ time where k is the output size. Therefore, we have

Theorem 3.3. *Given a polygon P and a diagonal e which cuts P into two parts, L and R, by using $O(n^2 \log n)$ time, we can construct a data structure of size $O(n^2)$ so that, for any query point $q \in R$, the partial visibility polygon $V_L(q)$ can be reported in $O(\log n + |V_L(q)|)$ time.*

3.2. *Computing Visibility Polygons by Balanced Triangulation*

In Section 3.1 we showed how to compute a partial visibility polygon. In this section we show how to combine it with a balanced triangulation to compute the entire visibility polygon.

The *balanced triangulation* of a simple polygon P was introduced by Chazelle [3]. The key observation is that there always exists a diagonal e of a simple polygon P that cuts P into two pieces, each with at most $2n/3$ vertices. By recursively subdividing each of the subregions resulting from cutting P along e, a balanced binary tree can be created where each interior node i corresponds to a subpolygon P_i and a diagonal e_i of P_i. The left and right subtrees of i correspond to two polygons, L_i, R_i, obtained by cutting P_i along e_i. The leaves are the triangles of the balanced triangulation. Denote by $\mathrm{lc}(i)$, $\mathrm{rc}(i)$, $\mathrm{pa}(i)$ i's left child, right child, and parent, respectively. Also assign the level of node i to the diagonal e_i (Fig. 8(a)).

In addition, for interior node i in the tree, we build a structure as described in Section 3.1 for reporting the partial visibility polygon in L_i and R_i with respect to the diagonal e_i. We also construct a point location structure on top of the triangulation. For simplicity of notation, let $V_i(q)$ denote $V_{P_i}(q)$.

Now, to compute $V(q)$ of a query point q, we first locate q among the leaf triangles. Let the path, from the leaf to the root, be i_1(leaf), $i_2, \ldots, i_k$(root) (Fig. 8(b)). We will see how to construct all the $V_i(q)$'s for i in the path inductively. For the leaf node i_1, $V_{i_1}(q)$ is simply the corresponding triangle. In the inductive step, suppose we have constructed $V_{i_j}(q)$, and without loss of generality, suppose $i_j = \mathrm{lc}(i_{j+1})$, i.e., i_j is the left child of i_{j+1}. We first compute $V_{\mathrm{rc}(i_{j+1})}(q)$ by querying the structure stored in the node i_{j+1}. If it is empty, we simply return $V_{i_{j+1}} = V_{i_j}$. Otherwise, we glue $V_{i_j}(q) = V_{\mathrm{lc}(i_{j+1})}(q)$ and $V_{\mathrm{rc}(i_{j+1})}(q)$ along the diagonal $e_{i_{j+1}}$ to obtain $V_{i_{j+1}}(q)$. To obtain efficient gluing, we represent each $V_i(q)$ in a circular list and store in an auxiliary array the pointers pointing to the diagonal edges appearing in $V_i(q)$. Since a partial visibility polygon has at most one diagonal edge of each level, we can simply keep the pointers indexed by their levels. To glue two partial visibility polygons, we first locate the diagonal edge e along which they possibly can be glued. This can be done in $O(1)$ time by a direct access to the auxiliary array. Then we split two circular lists by deleting the entries corresponding to

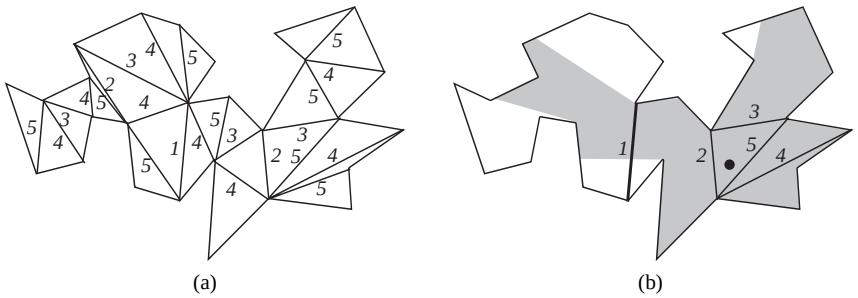


Fig. 8. (a) A balanced triangulation of P, where the number next to each edge is the level of that edge. (b) The procedure to glue partial visibility polygons together to obtain the visibility polygon. The diagonal edges shown in the figure are those edges on the path in the balanced triangulation.

$e_{i_{j_1}}$ and merge two lists together. We also need to update the auxiliary array. Since we know there is at most one edge from a level, it takes $O(\log n)$ time by simply copying the pointers from two previous arrays into the current one.

Thus, we have

Theorem 3.4. *A simple polygon P can be processed in $O(n^2 \log n)$ time into a data structure of size $O(n^2)$ so that, for any query point q, $V(q)$ can be reported in time $O(\log^2 n + |V(q)|)$.*

Proof. Note that in preprocessing, the space and time used by constructing exterior visibility decompositions dominate—all the other structures use a total of $O(n)$ space and $O(n \log n)$ preprocessing time.

The space and preprocessing time used to construct an individual exterior visibility decomposition for an m-sided polygon are $O(m^2)$ and $O(m^2 \log m)$, respectively. Thus, the space, denoted by $S(n)$, and preprocessing time, denoted by $T(n)$, used for an n-sided polygon in our algorithm satisfy the following recurrence:

$$S(n) = \max_{n/3 \le m \le 2n/3}(S(m) + S(n-m)) + \Theta(n^2),$$

$$T(n) = \max_{n/3 \le m \le 2n/3}(T(m) + T(n-m)) + \Theta(n^2 \log n).$$

Therefore, $S(n) = \Theta(n^2)$, and $T(n) = \Theta(n^2 \log n)$.

As for the query time, point location can be performed in $O(\log n)$ time. In addition, because the triangulation is balanced, the length of any path from the root to a leaf is $O(\log n)$. For each node i, the time needed to query the structure $V_i(q)$ is $O(\log n + |V_i(q)|)$ as shown in Theorem 3.3. Each merging can be done in $O(\log n)$ time according to the above discussion. Therefore, in total, the query time is $O(\log n + \sum_i (\log n + |V_i(q)|)) = O(\log^2 n + |V(q)|)$. □

Remark 3.1. In Section 3.1.1 we used the shortest-path query data structure to compute $C_e(q)$. It is unnecessary as in the above procedure, once we have computed $V_{i_j}(q)$, we know the visibility from q to $e_{i_{j+1}}$. This is because $e_{i_{j+1}}$ is the separating diagonal and on the boundary of both $L_{i_{j+1}}$ and $R_{i_{j+1}}$. If it is visible to q, then it must appear on the boundary of $V_{i_j}(q)$.

Since for a red-black tree, we can report the number of items inside any range in $O(\log n)$ time, the above algorithm can be modified to report the size of the visibility polygon of any query point in $O(\log^2 n)$ time.

Corollary 3.5. *A simple polygon P can be preprocessed into a data structure using $O(n^2)$ space and $O(n^2 \log n)$ time so that for any query point q, the size $|V(q)|$ of $V(q)$ can be reported in $O(\log^2 n)$ time.*

As another application, the above method can be extended to the cone visibility query problem. In a *cone visibility query problem*, in addition to a query point q, a query also includes a cone with q as the apex which delimits the visibility of q. We are asked to

report the visibility from q within the cone. This can be done in the same space and time bound as above—to answer a cone visibility query, we still compute partial visibility polygons and glue them together. The only difference is that we need to prune each exterior visibility polygon by the query cone $C(q)$ before gluing them together. We can first overlay two cones $C(q)$, $C_e(q)$ to obtain a single cone in $O(1)$ time and use this cone to perform a range search in the procedure described in Section 3.1.3.

Corollary 3.6. *Given a simple polygon P, we can process it into a data structure with $O(n^2)$ space and in $O(n^2 \log n)$ time so that, for any query point q and a cone $C(q)$, the visible region from q within the cone $C(q)$ can be reported in time $O(\log^2 n + k)$ where k is the output size.*

The algorithm that we have just described needs quadratic space. Although this can happen in the worst case, we may expect a lower complexity of the visibility decomposition in practice. There are many different ways to measure the complexity of a scene. For example, we can consider the maximum complexity of the visibility polygon. If every point in P can see at most A vertices, then we know that the number of constraint lines is $O(nA)$ as a constraint line can only be created by a pair of mutually visible vertices. This gives us an $O(n^2 A)$ bound on the complexity of the visibility decomposition. While this does not seem to help us to reduce the quadratic complexity of our algorithm, another measurement, the maximum number of intersections between any line segment inside P and critical constraints, can be used to measure the complexity of the algorithm.

Corollary 3.7. *If for any line segment inside P, it can intersect at most S critical constraints, then the space and preprocessing time are $O(nS)$ and $O(nS \log n)$, respectively.*

Proof. We focus on the complexity of constructing exterior visibility decompositions. Since any line segment inside P intersects at most S critical constraints, each exterior visibility decomposition is formed by at most S lines, the constraint lines that cut a specific diagonal edge.

Thus, we can replace n^2 by S^2 in the recurrence in the proof of Theorem 3.4 and stop recursion if n is smaller than S. It is easy to verify that the space needed is $O(nS)$, and the preprocessing time is $O(nS \log n)$. □

Remark 3.2. In the above corollary we used the strong condition that each line segment in P can intersect at most S constraint lines. However, we actually only require that this holds for diagonal edges. Is there a better characterization to capture this condition?

4. Maintaining Visibility from a Moving Viewpoint

In this section we present an algorithm to maintain the visibility from a moving viewpoint. Suppose that we have a point p inside the polygon P, and p moves along a line. We will present a data structure by which the visibility can be maintained correctly as time goes on. To be precise, we maintain the combinatorial structure of $V(p)$

which only changes at discrete points in time. Our goal is to detect all such critical times and update the combinatorial structure accordingly at those times. In addition, our algorithm maintains the visibility in an on-line fashion, namely, once the motion of the point changes, the data structure can be updated efficiently. In fact, our algorithm fits the framework of kinetic data structures in [1] very well and satisfies all the efficiency criteria proposed in that paper. Further, the algorithm achieves output sensitivity as, in the terminology of [1], all the events here are *external events* that change the structure.

We denote by $p(t)$ the position of p at time t. Then the combinatorial structure of $V(p(t))$ will generally change as time goes on. Denote by t^-, t^+ the time immediately before and after t, respectively. When we say that a structure *changes at time* t, what is meant is that it is different at t^- and t^+.

Remark 4.1. The point p moves on along a fixed line, but its velocity need not be constant. The results in this section are valid as long as this motion along the line is such that we are able to compute in constant time the first time when p reaches a specific point on the line. For example, p's position along the line could be a low-degree algebraic function of time t.

Remark 4.2. We assume that p never collides with the boundary of P, i.e., p always moves in the interior of P. In fact, while we are maintaining $V(p)$, it is straightforward to detect such collisions.

Remark 4.3. Again, we make the general position assumption. That is, no three vertices of P are collinear and p never moves on a line that passes through two polygon vertices although it may cross such a line.

4.1. *Combinatorial Changes of the Shortest-Path Tree*

Instead of maintaining the visibility polygon, we maintain the shortest-path decomposition of p. This is sufficient since, as we have noted, the visibility polygon is a cell in the shortest-path decomposition. Further, the shortest-path decomposition can be easily obtained from the shortest-path tree by extending tree edges. We therefore reduce the problem to the maintenance of the shortest-path tree. In the following we discuss how to maintain the shortest-path tree. However, the shortest-path decomposition can be maintained by the same method.

Recall that the shortest-path tree $T(p)$ is the tree rooted at p formed by taking the union of the shortest paths from p to every vertex of P (Fig. 4). We show how to maintain the shortest-path tree (resp. decomposition) and thus the visibility polygon. Define the *principal child* vertex (edge) of a non-root node v of $T(p)$ to be the vertex w (resp. edge $\overline{vw}$) of $T(p)$ among the children of v such that the angle formed by $\overrightarrow{vw}$ and $\overrightarrow{uv}$, where u is the parent of v, is the smallest among all such angles. This corresponds to clockwise and counterclockwise extensions of visibility edges in [8] depending on the direction in which the shortest path turns.

On the combinatorial changes of the shortest-path tree, we have the following characterization.

Lemma 4.1. *As p moves in P, $T(p)$ changes combinatorially at time t if and only if at time t, p is collinear with two vertices that are either consecutive children of p or one, say u, is a child of p and the other is the principal child of u at time t^-.*

Proof. Observe that $T(p)$ changes if and only if $\pi(p, w)$ changes combinatorially for some vertex w of P. Consider the shortest-path decomposition of w, $TD(w)$. For $\pi(p, w)$ to change at time t, p must be on a constraint of $TD(w)$, which is a critical constraint of P. Thus, we know that p is collinear with two vertices, say u, v, that are visible to it at time t. Without loss of generality, we assume that $u \in \overline{pv}$. If p can see both vertices at time t^-, then u, v are consecutive first-level vertices of $T(p)$ at time t^-. Otherwise, v is the principal child of u on $T(p)$.

On the other hand, we shall show that any such collinearity causes a change of $T(p)$. It is obvious that if p becomes collinear with a child u and u's principal child v, $T(p)$ changes at t because v will be properly visible to p at t^+, causing v to become a child of p (Fig. 9). When p is collinear with two consecutive children u, v, we will show that either u is on $\overline{pv}$ or v is on $\overline{pu}$. If this were not true, then p would be on $\overline{uv}$. Since $\overline{uv}$ is inside P, it is a diagonal of P and divides P into two polygons P_1, P_2. Since every point in a polygon can *properly* see at least three vertices, p must be able to see properly at least one vertex other than u, v, of P_1 and P_2, respectively. These two vertices would be p's children on $T(p)$ and separate u, v, contradicting the assumption that u, v are consecutive children of p on $T(p)$.

Now, assume that u is on $\overline{pv}$, then at t^+ the visibility of v to p will be blocked by u, causing v to be deleted from p and inserted as u's principal child (Fig. 9). □

The update of $T(p)$ from t^- to t^+ is quite straightforward according to the above proof. If p is collinear with a first-level vertex u and u's principal child v, then v becomes visible from p at t^+. In this case we cut v (and the subtree) from u and insert it as a child

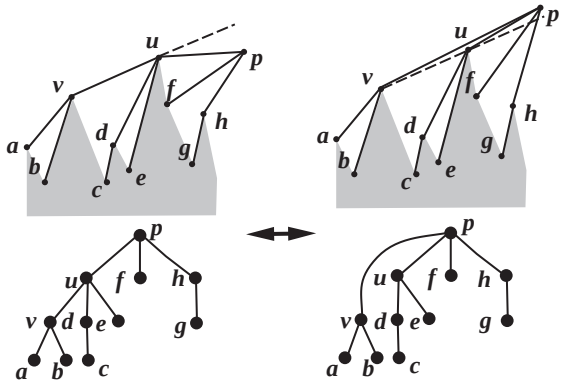


Fig. 9. Events in the maintenance of the shortest-path tree. In the first figure the principal children of u, v are v, a, respectively.

of p. If p is collinear with two consecutive first level vertices u, v, that means that one of those two vertices, say u, blocks the visibility from p to the other vertex v. In this case, v becomes the principal child of u at t^+ (Fig. 9).

4.2. *Tracking Shortest-Path Tree and Visibility Changes*

Notice that the number of children of p in $T(p)$ is $O(|V(p)|)$. As per Lemma 4.1, it is sufficient to check when the point p is collinear with $|V(p)|$ pairs of vertices to detect when $T(p)$ changes. The checking of collinearity is equivalent to detecting when p crosses the lines defined by those pairs. Assuming that p moves in a known manner, a simple solution is to maintain a priority queue in which we store the time when p crosses each of the constraint lines. Therefore, a change of $T(p)$ can be detected and processed in $O(\log|V(p)|)$ time if the motion of p is fixed. However, if p is permitted to change the line along which the motion occurs, we have to recompute the times when p crosses each constraint which would take $O(|V(p)|)$ time. To reduce this cost, we consider the convex face τ in the arrangement formed by all the constraint lines which contains the current point p. If we maintain the boundary of τ in a manner suitable for searching, for any change of p's motion, a binary search on the boundary of τ will tell us which edge is the next one p is going to cross provided p keeps moving along the current direction. The face τ can be maintained by a dynamic half-space intersection algorithm with $O(\log^2 n)$ cost per update. The algorithm is as follows.

For any P and a given initial position p, construct the shortest-path tree $T(p)$ from p to all vertices of P in linear time as in [10]. In additional linear time, we can obtain, for each vertex v, the doubly linked list of its children, sorted around v, with pointers to the first and last. Each vertex also stores a pointer to its principal child, which will always be either the first or the last one depending on how the shortest-path tree is bent locally.

By examining the first-level vertices of the shortest-path tree $T(p)$ and their principal children, we collect a set of $O(|V(p)|)$ lines that are defined by two consecutive children of p or by a child of p and its principal child, as per Lemma 4.1. Process the intersection of the half-spaces bounded by these lines and containing p, i.e., the face τ (Fig. 10), into a dynamic maintenance algorithm for half-spaces intersection as given by Overmars and van Leeuwen [15]. The structure can be initialized in $O(|V(p)|\log|V(p)|)$ time using

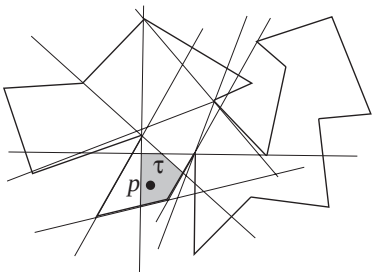


Fig. 10. Computing of the face τ. The thin lines in the figure are lines defined by first level vertices and their principal children. For clarity, the shortest-path tree is not drawn.

$O(|V(p)|)$ space and updated in $O(\log^2|V(p)|)$ time per insertion and deletion [15]. In $O(\log|V(p)|)$ time we can compute when the point p crosses the boundary of τ along the current direction as τ is a convex face.

After the above preprocessing, the shortest-path tree can be maintained by processing two types of events. One is when p crosses a constraint, i.e., the boundary of the face τ. This is when $T(p)$ changes. Depending on the way that p crosses the constraint, we update $T(p)$ as shown in Fig. 9. Namely, we either cut a principal child v from a first-level vertex u and insert v immediately before u as a first-level vertex or vice versa (Fig. 9). Because we store all the children of a vertex in a list in the order they appear on $T(p)$ and keep a pointer pointing to the principal child, these updates can be done in $O(1)$ time. Furthermore, we add and remove the appropriate constraints in the dynamic data structure representing the current face τ which contains p. Since there is a constant number of changes, this operation takes $O(\log^2|V(p)|)$ time by Overmars and van Leeuwen's algorithm. Finally, we compute the next time when p crosses the boundary of τ again in $O(\log|\tau|) = O(\log|V(p)|)$ time.

The other type of events is when the motion of p is updated. Since its motion is linear, we just need to perform a search to determine which edge, on the boundary of τ, p is going to cross next. This can be done again in $O(\log|V(p)|)$ time.

The above algorithm gives us a way to maintain the shortest-path tree of a moving point. If, for each tree edge vw, with v the parent of w, we also maintain the edge of P hit by $c(v, w)$, we can effectively maintain the shortest-path map in the same time bound.

To summarize, we have

Theorem 4.2. *Let P be a simple polygon and let p be a point in P. After $O(n \log n)$ time and $O(n)$ space preprocessing, if p undergoes linear motion, the time when the first combinatorial change happens in $T(p)$ (and $TD(p)$) can be determined in $O(\log|V(p)|)$ time. The data structure can be updated in time $O(\log^2|V(p)|)$ per change of $T(p)$ and in time $O(\log|V(p)|)$ per flight plan change.*

Since the visibility polygon of p is the same as the cell in $TD(p)$ that contains p, the above theorem implies that

Corollary 4.3. *For a simple polygon P and a point p moving inside P with a linear motion, the visibility polygon $V(p)$ can be maintained in $O(\log^2|V(p)|)$ time per change where a change might be either a combinatorial change of $V(p)$ or a flight plan change of the motion of p.*

5. Applications

In previous sections we propose algorithms for answering visibility queries and maintaining the visibility polygon from a moving viewpoint. In this section we exploit the connection between them and show some applications of their combination. We consider two problems. One is the space–query-time tradeoff in answering visibility queries. The other is the weak visibility query problem.

5.1. *Space–Query-Time Tradeoff in Answering Visibility Queries*

In this section we combine the results from Sections 3 and 4 to give an algorithm with a space–query-time tradeoff. Namely, we can guarantee query time $O((n^2/m)\log^3 n + |V(q)|)$ if $O(m)$ space is allowed for the data structure, which can be computed in $O(m \log n)$ time, where m is between $\Omega(n \log^3 n)$ and $O(n^2 \log n)$.

Notice that in the algorithm presented in Section 3, the space bottleneck comes from storing the entire exterior visibility decomposition, which may have size $\Theta(n^2)$. In what follows we focus on constructing and querying the exterior visibility decomposition. The storage and time bounds of the full algorithm then follow easily.

To obtain a tradeoff we compute a coarser decomposition, use it to determine visibility information for a nearby point, and then construct the answer to the query by walking from the nearby point to it while maintaining visibility during the walk.

A $(1/r)$-*cutting* of an arrangement of lines is the decomposition of the plane into r^2 triangular *cells* so that each cell is intersected by at most n/r lines. Such a decomposition always exists and can be constructed in $O(nr + n \log n)$ time [14], [4] for any r between 1 and n. Recall that the exterior visibility decomposition is an arrangement of n line segments created by clipping lines in a box B. We can thus construct a $(1/r)$-cutting of those lines and intersect the cells in the cutting with B. This way we obtain a cell decomposition $\mathcal{R}$ of B with r^2 cells and with each cell intersected by $O(n/r)$ critical constraints in $EV_L(q)$.

Now, in a manner similar to the algorithm computing $EV_L(q)$, by a depth-first traversal of the 1-skeleton of $\mathcal{R}$, we can obtain a path γ that visits every vertex of $\mathcal{R}$ and does not traverse any edge more than twice. The number of edges in γ is clearly $O(|\mathcal{R}|) = O(r^2)$. We compute and store visibility information $V(p)$, for all vertices p of $\mathcal{R}$, in a common persistent data structure, in the order of their appearance on γ. The space required is proportional to the sum of the number of differences between $V(p)$ and $V(p')$, for all pairs of adjacent vertices (p, p') on γ and, given the changes between consecutive vertices, the time required to build the structure is $O(\log n)$ times the storage requirements [16]. Actually, the structure we build is slightly different from that presented in [16] as we also require to be able to perform transient updates; see the discussion in Section 2.2.

When we traverse an edge $e = (q, q')$ on γ, since we know which constraints e cuts, we can compute all the intersections and sort them according to the order of crossing e. During the traversal, we update $V(\cdot)$ accordingly once a constraint is crossed, using the algorithm presented in Section 4. Since the number of constraints meeting the interior of a cell is n/r, the sorting procedure takes $O(n/r \log(n/r))$ time, and, during the transversal, there are at most n/r updates, each resulting in $O(1)$ changes in $V(\cdot)$, at the price of $O(\log n)$ per change. Thus collecting all the lists $V(p)$ and building the persistent data structure requires

$$|\mathcal{R}| \cdot O(n/r) \cdot O(\log n) = O(rn \log n)$$

time and $O(rn)$ space. In addition, we preprocess the decomposition $\mathcal{R}$ into a point-location structure. This can be done in $O(|\mathcal{R}| \log |\mathcal{R}|) = O(r^2 \log r)$ time, using standard point-location algorithms (e.g., [16]).

After the above structures are built, a query is answered by looking up the cell Δ of $\mathcal{R}$ containing q. We then pick any vertex p of Δ and retrieve the entry of $V(p)$ stored

in the persistent data structure. Then, compute the intersections between $\overline{pq}$ and the constraints cutting Δ, and trace the segment $\overline{pq}$ using the on-line algorithm. The tracing requires at most n/r updates where each update is a transient update, so it can be done in $O((n/r)\log n)$ time as per Lemma 2.3. As noted in Section 3.1.2, the resulting structure should be suitable for searching in $V(q)$, without explicitly exhibiting it, to obtain output sensitivity. Since the visibility polygons are stored in a persistent red-black tree allowing transient updates, the structure we obtained satisfies the requirement.

Thus for computing the partial visibility polygon, preprocessing time is $O(rn\log n)$, space is $O(rn)$, and query time is $O((n/r)\log n)$. As for the entire algorithm, plugging these bounds into the recurrence in the proof of Theorem 3.4, we have that the preprocessing time is $O(rn\log^2 n)$, space is $O(rn\log n)$, and query time is $O((n/r)\log^2 n)$. Now recall that r can be chosen between 1 and n, so putting $m = rn\log n$, we obtain:

Theorem 5.1. *Given a simple n-gon in the plane and a number m between $n\log^3 n$ and $n^2\log n$, one can preprocess the polygon in $O(m\log n)$ time and $O(m)$ space so that, for a query point q, $V(q)$ can be computed in $O((n^2/m)\log^3 n + |V(q)|)$ time.*

Remark 5.1. We choose m to be $\Omega(n\log^3 n)$ because the overhead should be at most linear to be interesting.

5.2. *Answering Weak Visibility Queries*

In this section we show the application of the combination of visibility query and maintenance algorithms developed in the previous sections to the weak visibility query problem. In the weak visibility query problem, we are given a simple polygon P and asked to build a data structure to return the weak visibility polygon $V(s)$ for any query segment s in P. There are algorithms solving this problem in linear time without preprocessing. Here, again our goal is to report the weak visibility polygons in output-sensitive time.

To compute $V(s)$, imagine that there is a point moving along s from one to the other endpoint with constant velocity. Once we can maintain the visibility polygon from this moving viewpoint, we can compute $V(s)$ because $V(s)$ is the union of all the visibility polygons of the points on s. First note that we cannot copy the algorithm in Section 4 directly because we cannot afford the initialization cost of building the shortest-path tree, which may take linear time. However, we needed the shortest-path tree in order to compute the constraints in Lemma 4.1. In the following we show another method for computing those constraints without constructing the shortest-path tree.

Recall that there are two types of constraints. Constraints of the first type are created by two adjacent vertices in the visibility polygon. The others are created by a vertex on the visibility polygon and its principal child in the shortest-path tree. It is straightforward to compute the constraints of the former type once we have the visibility polygon. For the latter ones, we need an efficient way to compute the principal child, w, of a vertex $v \in V(p)$ on the shortest-path tree $T(p)$. Consider the circular polar order on all the vertices with respect to the vertex v. According to the definition, the principal child w of v is the vertex in $V(v)$ so that w is the rightmost/leftmost vertex on the left/right side of the extension of $\overline{pv}$—which side to consider depends on how the shortest path turns

at v. In the following, without loss of generality, we assume that the principal child is the rightmost vertex on the left side of the extension of $\overline{pv}$.

As in Section 3.2, we decompose the polygon P into canonical pieces. For a canonical polygon Q, call the rightmost vertex in $V_Q(v)$ on the left side of the extension of $\overline{pv}$ the *candidate* principal child of p. Then the principal child is the one closest to the extension of $\overline{pv}$ among all the $O(\log n)$ candidate principal children. To find the candidate principal child for a subpolygon Q, we follow the method presented in Section 3.1.3. Instead of querying the exterior visibility polygon by a range, we can query it by "finding the element closest to a query *key* on the appropriate *side* within a query *range*," where the query key corresponds to the extension of $\overline{pv}$, the side corresponds to which side we would like to search, and the query range corresponds to the visibility cone in Section 3.1.1. This gives us the candidate principal child in that subpolygon in $O(\log n)$ time. Thus, the principal child can be computed in $O(\log^2 n)$ time once we have the structure built in Section 3.

Combining with the maintenance part in Section 4.2, we can still maintain the visibility polygon of a moving viewpoint in $O(\log^2 n)$ time per event after $O(|V(p)|\log^2 n)$ initialization cost.

Regarding the number of events that occur while p traverses a segment, we have the following lemma.

Lemma 5.2. *The number of events that occur while the viewpoint moves from one endpoint of segment $s \subset P$ to the other is $O(|V(s)|)$.*

Proof. For any vertex v on ∂P, the portion of s visible to v is a single subsegment of s if P is simple. In other words, if a viewpoint p moves from one endpoint of s to the other, it can see v only when it is in a contiguous interval of s. Therefore, once a vertex disappears from $V(p)$ during the traversal, it will never come to be visible to p again. Since each event must cause a vertex to appear or disappear from $V(p)$, we charge that event to that vertex in both cases. There are $O(|V(s)|)$ vertices which can be charged, and each vertex can be charged at most twice. Thus, the total number of events is $O(|V(s)|)$. □

Therefore, for a point p moving along the segment s, the total number of changes of $V(p)$ is $O(|V(s)|)$, and we can detect each change in $O(\log^2 n)$ time and update the data structure in $O(\log^2 n)$ time per change. As in Section 4.2, we can update the combinatorial description of the weak visibility polygon accordingly once $V(p)$ changes.

There is one more issue on how to compute $V(s)$ from its combinatorial description. Unlike the visibility polygon of a point, $V(s)$ is not star-shaped. Therefore, for an edge $e \in V(s)$, we need additional information to compute the portion of e that is weakly visible to s. Since that portion is a subsegment of e, it suffices if we know how to compute its endpoints, i.e., the extreme points on e that are weakly visible to s. Observe that during the above traversal, we know when e starts or ceases to be visible from the imaginary moving point, i.e., we know the endpoints of the subsegment of s that is weakly visible to e. By the following lemma, it suffices to consider the visibility from e to these two endpoints, which can be done in $O(\log n)$ time as in Section 3.1.1.

Lemma 5.3. *For two segments s_1, s_2 inside a simple polygon P, denote by b_1 (b_2) the set of points on s_1 (s_2) weakly visible to s_2 (s_1). Then b_1, b_2 are either empty sets or*

line segments. Further, each endpoint of b_1 is either an endpoint of s_1 or visible to an endpoint of b_2 and vice versa.

Proof. By simplicity of P, b_1, b_2 are connected sets. Thus, they are either empty sets or line segments.

We observe that an interior point p of s_1 can be an endpoint of b_1 if and only if it sees exactly one point on s_2. Further, if $q \in s_2$ is the only point visible to p, then p is also the only point visible to q. These two facts imply the above lemma. □

Thus, we obtain that

Theorem 5.4. *Given a simple n-gon P in the plane, one can preprocess P in $O(n^2)$ space and $O(n^2 \log n)$ time so that, for any segment s inside P, $V(s)$ can be computed in $O(|V(s)| \log^2 n)$ time.*

6. Conclusion

In this paper we showed how to answer visibility queries in nearly optimal time by using quadratic preprocessing time and storage, which improves on the previous algorithms by a linear factor. Then we provided an algorithm which can maintain a visibility polygon from a point in linear space and $O(\log^2 n)$ time per change in visibility or flight plan.

A nice property of our method of answering visibility queries is that the visibility polygons are represented in a small number of canonical parts, each stored in a form suitable for searching. By exploiting this fact, we obtained algorithms for some other visibility problems. Later, we showed how to combine the results to yield a tradeoff for visibility queries and an output-sensitive algorithm for the weak visibility query problem.

An immediate open question is whether we can improve the space and preprocessing time further to a nearly linear bound while keeping the query time the same, i.e., a polylogarithmic additive overhead and $O(1)$ cost for each output. It is also interesting to find applications of our structures to other related problems, e.g. answering the query about the area of the visibility polygons. As to the maintenance problem, the bound relies on the fact that the motion is linear. The situation when the motion follows a general algebraic curve remains open. For both problems, it is interesting to know if our techniques can be extended to the case of polygons with holes.

In this paper our complexity bounds are worst-case bounds. However, in reality, the polygons that we deal with normally have less complex visibility decomposition than in the worst case. It would be interesting to know if we can design algorithms whose complexity depends on certain natural parameters. For example, two reasonable measures could be the maximum or the average number of vertices that any point can see.

Acknowledgment

The authors thank the anonymous referees for their comments leading to the improvement of the paper.

References

1. J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th ACM–SIAM Sympos. Discrete Algorithms*, pages 747–756, 1997.
2. P. Bose, A. Lubiw, and J. I. Munro. Efficient visibility queries in simple polygons. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 23–28, 1992.
3. B. Chazelle. A theorem on polygon cutting with applications. In *Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 339–349, 1982.
4. B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9(2):145–158, 1993.
5. B. Chazelle and L. J. Guibas. Visibility and intersection problems in plane geometry. In *Proc. 1st Annu. ACM Sympos. Comput. Geom.*, pages 135–146, 1985.
6. D. Z. Chen and O. Daescu. Maintaining visibility of a polygon with a moving point of view. In *Proc. 8th Canad. Conf. Comput. Geom.*, pages 240–245, 1996.
7. H. ElGindy and D. Avis. A linear algorithm for computing the visibility polygon from a point. *J. Algorithms*, 2:186–197, 1981.
8. S. K. Ghosh and D. M. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Comput.*, 20:888–910, 1991.
9. L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *J. Comput. System Sci.*, 39:126–152, 1989.
10. L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.
11. L. J. Guibas, R. Motwani, and P. Raghavan. The robot localization problem in two dimensions. In *Proc. 3rd ACM–SIAM Sympos. Discrete Algorithms*, pages 259–268, 1992.
12. J. Hershberger and S. Suri. A pedestrian approach to ray shooting: shoot a ray, take a walk. *J. Algorithms*, 18:403–431, 1995.
13. D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.
14. J. Matoušek. Construction of ε-nets. *Discrete Comput. Geom.*, 5:427–448, 1990.
15. M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. System. Sci.*, 23:166–204, 1981.
16. N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Comm. ACM*, 29:669–679, 1986.
17. G. Vegter. The visibility diagram: a data structure for visibility problems and motion planning. In *Proc. 2nd Scand. Workshop Algorithm Theory*, volume 447 of Lecture Notes in Computer Science, pages 97–110. Springer-Verlag, Berlin, 1990.

Received June 12, 2000, *and in revised form August* 27, 2001. *Online publication March* 27, 2002.