TR953

# An Optimal Algorithm for Computing Visibility in the Plane

(Extended Abstract)

Paul J. Heffernan*
Joseph S. B. Mitchell†
SORIE, Cornell University, Ithaca, NY 14853

December 5, 1990

**Abstract**

We give an algorithm to compute the visibility polygon from a point among a set of $h$ pairwise-disjoint polygonal obstacles with a total of $n$ vertices. Our algorithm uses $O(n)$ space and runs in optimal time $\Theta(n + h \log h)$, improving the previous upper bound of $O(n \log h)$.

## 1 Introduction

Let $\mathcal{D}$ be a planar polygonal domain with $h$ holes and $n$ vertices: $\mathcal{D}$ is a connected closed subset of the plane whose boundary consists of a set of $n$ line segments. If $h = 0$, then $\mathcal{D}$ is simply connected and is called a *simple polygon*. If $h > 0$, then $\mathcal{D}$ is multiply connected, and its holes form a set $\mathcal{P} = \{P_1, P_2, \ldots, P_h\}$ of pairwise-disjoint simple polygons in the plane. The *visibility polygon* with respect to a point $s \in \mathcal{D}$ is the locus of all points $q \in \mathcal{D}$ such that $\overline{sq} \subset \mathcal{D}$. The problem of computing the visibility polygon with respect to a given point $s$ is known as the "hidden line removal" problem and is fundamental in computational geometry.

**Our Result.** We provide an algorithm to compute a visibility polygon in optimal time $\Theta(n + h \log h)$ and space $O(n)$.

*Email: heff@orie.cornell.edu. Supported by an NSF graduate fellowship.

†Email: jsbm@cs.cornell.edu. Partially supported by a grant from Hughes Research Laboratories, Malibu, CA, and by NSF Grant ECSE-8857642.

**Relation to Previous Work.** Algorithms to compute the visibility polygon have been known for some time; a clear summary of the many known visibility algorithms is given in chapter 8 of O'Rourke's book [O'R]. For the case of a simple polygon $P$, optimal $O(n)$ algorithms have been given by [EA,Le2] and by [JS], who correct a minor error in [EA,Le2] while simplifying the algorithm of [Le2]. For the case of a polygon $P$ with holes, straightforward $O(n \log n)$ time algorithms can be based on plane (rotational) sweep about $s$ (as in [Le1,SO]) or based on divide-and-conquer (as in [AM]). In fact, by using the linear-time algorithms for simple polygons to compute the visible portion of the boundary of each hole, and then merging these "profiles", one can obtain a simple $O(n \log h)$ algorithm for computing the visibility polygon (see [AM,AAGHI]).

There is an $\Omega(n + h \log h)$ lower bound (from sorting) for computing a visibility polygon [AAGHI,O'R,SO]. Optimal algorithms that achieve this time bound were known for the special case in which the holes $P_i$ are convex ([AM,AAGHI]) or star-shaped ([AM]). The question of whether or not an algorithm exists for the *general* case whose running time is *linear* in $n$ has been a fundamental open problem. No algorithm was previously known with running time $O(n + f(h))$ for *any* function $f(h)$.

We resolve the open question by providing a few different methods, culminating in an optimal algorithm. We outline our approach below. We let $\tau(n)$ denote the time required to triangulate a simple polygon. By Chazelle's recent breakthrough [Ch2], we know that $\tau(n) = O(n)$, and there are several fast deterministic and randomized algorithms giving bounds of $O(n \log \log n)$ ([KKT,TV]) or $O(n \log^* n)$ ([Ch1,CTV,Se]).

(1) We give a very simple $O(n + h^2)$ algorithm.

(2) We have an $O(\tau(n) + h \log^2 h)$ algorithm that is relatively simple, but relies on triangulation. This algorithm is not discussed in this abstract, since it does not directly lead to our strongest theoretical result.

(3) We give an $O(\tau(n) + h \log(\bar{n} + h))$ algorithm for the special case in which all of the holes are *stabbed* by a line. Here, $\bar{n} \leq n$ is the number of sides of the most complex hole.

(4) We give an $O(\tau(n) + h \log \log h \log^2 \bar{n})$ algorithm based on applying the result (3) to $O(\log h)$ classes of obstacles, and then merging the resulting set of visibility polygons.

(5) Finally, we show how the algorithm of (4) can be modified to yield an optimal time bound of $\Theta(n + h \log h)$ for the general problem.

# 2 Notation and Basic Properties

The *visibility profile* of a set of pairwise-disjoint polygonal obstacles $P_1, \ldots, P_h$ is the "lower envelope" of the obstacles — it is what is seen by an observer at $y = -\infty$. In order to solve our problem of computing the visibility polygon from a point $s \in \mathcal{D}$, it suffices to

consider the problem of computing the visibility profile of the set of holes of $\mathcal{D}$. In $O(n)$ time, we can compute the visibility polygon with respect to $s$ in the simple polygon whose boundary is the outer boundary of $\mathcal{D}$, and then merge this with the visibility profile of the obstacles (holes of $\mathcal{D}$) viewed in the polar coordinate system $(\theta, r)$ centered at $s$. Thus, without loss of generality we concentrate on the problem of finding the visibility profile of the polygons $P_1, \ldots, P_h$.

We let $x(p)$ and $y(p)$ denote the $x$- and $y$-coordinates of a point $p$ in the plane. If a value $x$ represents an $x$-coordinate, then let $x^-$ and $x^+$ denote the $x$-coordinate values infinitesimally left and right of $x$, respectively.

A polygon $P_i$ contains two vertices, $l_i$ and $r_i$, of minimum and maximum $x$-coordinate, respectively. Since we assume that the observer is at $y = -\infty$, the chain obtained by traversing $P_i$ clockwise from $l_i$ to $r_i$ is completely blocked from the view of the observer by the counterclockwise chain of $P_i$ from $l_i$ to $r_i$. We therefore use only this lower chain of $P_i$ when computing the visibility profile. In fact, in the remainder of this paper, we assume that $P_1, \ldots, P_h$ are polygonal chains joining their left endpoints ($l_i$) to their right endpoints ($r_i$).

For any set $S \subseteq \mathcal{P} = \{P_1, \ldots, P_h\}$, we let $VP(S)$ denote the visibility profile of the chains in $S$, and we let $VP(S; x)$ denote the point of $VP(S)$ with $x$-coordinate $x$. We abuse notation slightly and write $VP(i, \ldots, j)$ and $VP(i, \ldots, j; x)$, instead of $VP(\{P_i, \ldots, P_j\})$ and $VP(\{P_i, \ldots, P_j\}; x)$.

We can think of the profile $VP(S)$ as a piecewise-continuous function over the domain $[x(l_{min}), x(r_{max})]$, where $x(l_{min}) = \min_{i \in S} x(l_i)$ and $x(r_{max}) = \max_{i \in S} x(r_i)$. The points $x$ of discontinuity of $VP(S)$ are of two types:

- $x$ is a *jump* if $VP(S)$ coincides with the same chain $P_i$ at both $x^-$ and $x^+$; and
- $x$ is a *leap* if $VP(S)$ coincides with distinct chains $P_i$ and $P_j$ $(i \neq j)$ at $x^-$ and $x^+$.

A maximal connected subdomain of $[x(l_{min}), x(r_{max})]$ not containing a leap in its interior is called a *piece* (the corresponding section of $VP(S)$ over this domain is also called a piece). Since a piece of $VP(S)$ corresponds to a section of a specific chain $P_i$, we say that $P_i$ *appears* in $VP(S)$ with this piece. These definitions are illustrated in Figure 1.

While we have thought of $VP(S)$ as a function in order to define jumps and leaps, our algorithms will store a visibility profile $VP(S)$ as a polygonal chain. A vertical edge of the chain $VP(S)$ corresponds to a jump or leap. We call the vertical edge of a jump of a profile $VP(i)$ a *lid*, since its interior is disjoint from $P_i$. We will usually represent a lid by $\overline{ab}$, where $y(a) < y(b)$; therefore, if $\overline{ab}$ is a lid of $VP(i)$, then $VP(i; x(a)) = a$.

A leap $x$ between chains $P_i$ and $P_j$ can occur in one of two manners. The leap $x$ may be caused by the left or right endpoint of one of the two chains, or it may occur where one profile intersects a lid of the other. These cases are illustrated in Figure 1: Coordinates $x_1$ and $x_3$ correspond to leaps at a left endpoint, $x_5$ and $x_6$ correspond to leaps at a right endpoint, and $x_2$ and $x_4$ correspond to leaps at lids.

We will often use the expression "$p$ is below $q$" to indicate that $y(p) < y(q)$. Similar use is made of the terms "above", "left", and "right". We say that profile $VP(S)$ is below

profile $VP(S')$ at $x$-coordinate $x$ if $y(VP(S;x^-)) < y(VP(S';x^-))$ *or* if $y(VP(S;x^+)) < y(VP(S';x^+))$. It is possible for one but not both of these conditions to hold if one of the profiles has a jump or leap at $x$.

We will assume without loss of generality that all chains $P_1, \ldots, P_h$ lie completely above the $x$-axis, so that the point $p = (x, 0)$ is below $VP(i)$ for any $x$ and any profile $VP(i)$.

**Lemma 1** *If $x(l_i) < x(l_j)$, then $P_j$ appears at most once in $VP(i,j)$; that is, at most one piece of $VP(i,j)$ is contributed by $VP(j)$.*

**Proof.** Suppose that $VP(j)$ contributes two pieces to $VP(i,j)$. Let $p$ and $q$ be points of $VP(j)$ on each of the two pieces, with $p$ on the left piece and $q$ on the right piece. Let $r$ be a point of $VP(i)$ that lies on a piece between the two pieces contributed by $VP(j)$, so that $x(p) < x(r) < x(q)$. Refer to Figure 2.

Now consider the closed Jordan curve given by starting at point $(x(p), 0)$, going up to $p$, following chain $P_j$ to $q$, going down to $(x(q), 0)$, and then returning to $(x(p), 0)$ along the $x$-axis. Since point $r$ is on the profile $VP(i,j)$, it must lie in the bounded component defined by this Jordan curve. On the other hand, the left-most point $l_i$ must lie in the unbounded component defined by the closed curve, since $l_i$ is to the left of $l_j$. This implies that $P_i$ must cross the Jordan curve, which is a contradiction, since $p$ and $q$ are on the profile, and $P_i$ and $P_j$ do not cross. □

**Lemma 2** *If $x(r_j) < x(r_i)$, then $P_j$ appears at most once in $VP(i,j)$.*

**Proof.** Similar to Lemma 1. □

We can now give a full characterization of $VP(i,j)$, for chains $P_i$ and $P_j$ whose $x$-coordinate domains overlap. Assume without loss of generality that $x(l_i) < x(l_j)$. Refer to Figure 3.

(1) If $x(r_i) < x(r_j)$, then clearly $P_i$ and $P_j$ each appear at least once in $VP(i,j)$, and by the previous two lemmas, each appears at most once. The profile $VP(i,j)$ therefore consists of a piece from $VP(i)$ lying left of a piece from $VP(j)$.

(2) If $x(r_j) < x(r_i)$, there are two possibilities:

   (a). Profile $VP(j)$ may lie completely above $VP(i)$, in which case $VP(i,j)$ is the single piece $VP(i)$.

   (b). If $P_j$ appears once in $VP(i,j)$, then $P_i$ must appear exactly twice, since pieces alternate, and the left- and right-most pieces are from $P_i$.

We now state a combinatorial lemma of fundamental importance:

**Lemma 3** *The profile $VP(S)$ has $O(|S|)$ pieces.*

**Proof.** Consider the (ordered) sequence $\sigma$ of indices of chains $P_i$ that contribute pieces to $VP(S)$. There are $|S|$ different indices, and by the definition of pieces, no index $i$ can appear twice consecutively in $\sigma$. By Lemma 1 (or Lemma 2), it is not possible to have a subsequence of the form $\ldots, i, \ldots, j, \ldots, i, \ldots, j, \ldots$. This implies that $\sigma$ is a Davenport-Schinzel sequence of order 2, so its maximum length is given by $\lambda_2(|S|) = 2|S| - 1$. (See [Sh] for background on the theory of Davenport-Schinzel sequences.) □

# 3 An $O(n + h^2)$ Algorithm

We describe now a simple $O(n+h^2)$-time algorithm for computing the visibility profile of a collection of disjoint polygons. Not only is the algorithm relatively easy to implement, but it resolves the previously open theoretical question of whether or not an algorithm *linear* in $n$ exists.

Assume that we have indexed the chains $\mathcal{P} = \{P_1, \ldots, P_h\}$ so that their left endpoints $l_1, \ldots, l_h$ are sorted by decreasing $x$-coordinate. The algorithm simply considers the profiles one-by-one according to this order: Step $i$ consists of adding $VP(i)$ to $VP(1, \ldots, i-1)$ to obtain $VP(1, \ldots, i)$. The time to update the profile when we insert $VP(i)$ is linear in $h$ and the size of $P_i$, implying the claimed overall time bound.

The algorithm maintains a sorted list of the leaps, $x_1, \ldots, x_K$, of the current profile $VP(1, \ldots, i-1)$. Each leap $x_k$ in the list stores a pointer to the point $VP(1, \ldots, i-1; x_k)$. To add $VP(i)$ to the profile, we traverse $VP(i)$ to place pointers on the points $VP(i; x_1), \ldots, VP(i; x_K)$. Now, for each leap $x_k$, we compare the points $VP(1, \ldots, i-1; x_k)$ and $VP(i; x_k)$, to determine whether $VP(i)$ is below $VP(1, \ldots, i-1)$ at this $x$-coordinate. If so, we have identified a piece of $VP(i)$ in $VP(1, \ldots, i)$; we simultaneously traverse $VP(i)$ and $VP(1, \ldots, i-1)$ to the left, maintaining our pointers at the same approximate $x$-coordinate, until we reach the $x$-coordinate, $x_l$, where $VP(i)$ is no longer below $VP(1, \ldots, i-1)$; $x_l$ is the left endpoint of this piece of $VP(i)$, and consequently is a leap in $VP(1, \ldots, i)$. Similarly, we simultaneously traverse $VP(i)$ and $VP(1, \ldots, i-1)$ to the right to obtain the right endpoint, $x_r$. The portion of $VP(1, \ldots, i-1)$ between $x_l$ and $x_r$ is replaced by the corresponding portion of $VP(i)$, and the leaps at $x_l$ and $x_r$ are incorporated into the list, along with pointers to $VP(1, \ldots, i; x_l)$ and $VP(1, \ldots, i; x_r)$. Of course, the interval $[x_l, x_r]$ may contain leaps of $VP(1, \ldots, i-1)$ other than $x_k$, but this poses no difficulty to the algorithm. The following lemma establishes that the new profile obtained in this manner is in fact $VP(1, \ldots, i)$, and that the updated list of leaps is the list for $VP(1, \ldots, i)$.

**Lemma 4** *Each piece of $VP(i)$ in $VP(1, \ldots, i)$ must cover a leap of $VP(1, \ldots, i-1)$; that is, for each piece contributed by $VP(i)$, there exists a leap $x$ of $VP(1, \ldots, i-1)$ such that $VP(1, \ldots, i; x) \in VP(i)$.*

**Proof.** Suppose we have a piece $[x_l, x_r]$ of $VP(i)$ in $VP(1, \ldots, i)$ that lies between consecutive leaps $x_k$ and $x_{k+1}$ of $VP(1, \ldots, i-1)$. Since $[x_k, x_{k+1}]$ is a single piece of

$VP(1,\ldots,i-1)$, the points $VP(1,\ldots,i-1;x_k^+)$ and $VP(1,\ldots,i-1;x_{k+1}^-)$ lie on the same profile $VP(j)$. We have that $VP(j)$ lies below $VP(i)$ at $x_k^+$ and $x_{k+1}^-$, and that $VP(i)$ lies below $VP(j)$ at $x_l^+$, where $x_k < x_l < x_{k+1}$. But this contradicts Lemma 1, since $x(l_i) < x(l_j)$, by our ordering of the polygonal chains. □

We now analyze the time complexity of the algorithm. The initial indexing of the polygonal chains requires time $O(h \log h)$, to sort the left endpoints of the chains. Adding $VP(i)$ to $VP(1,\ldots,i-1)$ requires that we traverse $VP(i)$ twice — once to place pointers to $VP(i;x_1),\ldots,VP(i;x_K)$, and once during the simultaneous traversals of $VP(i)$ and $VP(1,\ldots,i-1)$. The time spent in all steps except the traversing of $VP(1,\ldots,i-1)$ is $O(|P_i|+h)$, implying a total of $O(n+h^2)$ over all steps. The simultaneous traversals of the updating step require that we traverse sections of $VP(1,\ldots,i-1)$, which consists of profiles that have already been processed. However, the portions we traverse are deleted from the current profile $VP(1,\ldots,i)$, and are never traversed again. Thus, the entire algorithm runs in time $O(n+h^2)$.

# 4 An Optimal Algorithm

We turn our attention now to a different algorithm, one which attains the optimal $\Theta(n + h \log h)$ time bound. We will describe first an algorithm that runs in time $O(n+h \log\log h \log^2 n)$, and will then modify it to perform in optimal time.

We begin by sorting the $x$-coordinates of the endpoints of $P_1,\ldots,P_h$, thereby obtaining a list $x_1,\ldots,x_{2h}$. If $\mathcal{P} = \{P_1,\ldots,P_h\}$, define $S_1$ to be the chains of $\mathcal{P}$ stabbed by the vertical line $x = x_h$. Define $S_2$ to be the chains of $\mathcal{P} \setminus S_1$ stabbed by $x = x_{\lfloor h/2 \rfloor}$ or $x = x_{\lfloor 3h/2 \rfloor}$. Continuing in this way, we obtain a partitioning of $\mathcal{P}$ into a class of subsets $S_1,\ldots,S_{\lceil \log h \rceil}$. Below we will show that the visibility profile of a set $S'$ of $h'$ polygonal chains stabbed by a vertical line can be computed in time $O(n' + h' \log \bar{n}')$, where $n'$ is the total number of vertices in $S'$, and $\bar{n}'$ is the number of vertices on the largest chain in $S'$. Therefore $VP(S_1),\ldots,VP(S_{\lceil \log h \rceil})$ can be computed in time $O(n + h \log \bar{n})$, where $\bar{n}$ is the size of the largest chain in $\mathcal{P} = \{P_1,\ldots,P_h\}$. We will also show how to merge $VP(\mathcal{S}')$ with $VP(\mathcal{S}'')$ in $O(\bar{h} \log^2 \bar{n})$ time, for two subsets $\mathcal{S}'$ and $\mathcal{S}''$ of $\{S_1,\ldots,S_{\lceil \log h \rceil}\}$, with $max\{i|S_i \in \mathcal{S}'\} < min\{j|S_j \in \mathcal{S}''\}$, where $\bar{h}$ is the total number of polygonal chains in the sets comprising $\mathcal{S}'$ and $\mathcal{S}''$. This allows one to compute $VP(\mathcal{P})$ by recursively computing $VP(S_1 \cup \ldots \cup S_{\lceil \log h/2 \rceil})$ and $VP(S_{\lceil \log h/2 \rceil + 1} \cup \ldots \cup S_{\lceil \log h \rceil})$ and then merging them. Each step of the recursion requires time $O(h \log^2 \bar{n})$, and the recursion has depth $O(\log\log h)$, giving a total algorithm run-time of $O(n + h \log\log h \log^2 \bar{n})$.

Throughout our algorithm, we will wish to make logarithmic-time queries that we call *lid queries*. Basically, given a chain $P_i$ and a point in the plane $p$, a lid query of $p$ on $P_i$ asks where $p$ is with respect to $P_i$. The result of such a query will give us important information about $VP(\{P_i, c\})$, for any chain $c$ containing $p$ such that $c \cap P_i = \emptyset$. We now describe lid queries in detail.

Consider a lid $\overline{ab}$ of $VP(i)$, the visibility profile of chain $P_i$. The lid $\overline{ab}$ and the subchain

of $P_i$ between $a$ and $b$ form a simple polygon, which we call a *pocket*, such that all points in the interior of the pocket are non-visible from below. If a point $p$ in the pocket lies on a chain $c$ that does not intersect $P_i$, then $c$ can be below $VP(i)$ only if it crosses $\overline{ab}$. Since $\overline{ab}$ is on $VP(i)$, crossing $\overline{ab}$ is also a sufficient condition for $c$ to be below $VP(i)$ somewhere. We now state formally the information that we want from a *lid query*:

**Definition 1 (Lid Query)** *For a point $p$ not on $P_i$, one of the following is true:*

1. *$p$ is below $VP(i)$,*
2. *$p$ is in a pocket of $VP(i)$,*
3. *$p$ lies in the region above the simple, infinite chain $c = \rho_u(l_i) \cup P_i \cup \rho_u(r_i)$.*

*If (1) is true, a lid query of $p$ on $P_i$ returns $VP(i; x(p))$ (in Figure 4, for example, a query on $p_1$ returns $VP(i; x(p_1))$). If (2) is true, the lid query returns the lid $\overline{ab}$ that defines the pocket (e.g. a query on $p_2$ in the figure returns lid $\overline{a_2b_2}$). If (3) is true (as it is for $p_3$ in the figure), then the rays $\rho_u(l_i)$ and $\rho_u(r_i)$ together have the property that we desire in the lids, so the lid query returns them.*

Lid queries will be used often in our algorithm to determine quickly if and where a chain $c$ containing a point $p$ lies below the current profile $VP(S)$. Often, we will try to add a chain $c$ to $VP(S)$ to form $VP(S \cup \{c\})$, knowing only that $c$ contains a certain point $p$ and that $c$ and $S$ are disjoint. We formalize this notion by defining the *lid property*:

**Definition 2 (Lid Property)** *An $x$-coordinate $x$ has the lid property for point $p$ and profile $VP(S)$ if, for any chain $c$ that contains $p$ and is disjoint from $S$, $VP(\{c\})$ is below $VP(S)$ somewhere only if $VP(\{c\})$ is below $VP(S)$ at $x$.*

When a lid query of a point $p$ on a chain $P_i$ returns a lid $\overline{ab}$, the $x$-coordinate $x(a)$ satisfies the lid property for $p$ and $VP(i)$. Our algorithm will produce, through the use of lid queries, $x$-coordinates that satisfy the lid property for the current profile $VP(S)$; typically these will be the $x$-coordinates of either leaps or jumps of $VP(S)$.

Lid queries are basically planar point location. We first obtain the vertical visibility map of $P_i$ (in $O(n)$ time for all chains, by [Ch2]). Then $P_i$ can be preprocessed to return the trapezoid of the map containing a query point in logarithmic time (by [Ki], for example); this is sufficient to answer the query for case (1). For cases (2) and (3), we need to do some more work. Consider the dual graph of the trapezoidal decomposition, with the edge corresponding to the decomposition ray $\rho_u(r_i)$ deleted. This graph is a tree, and the trapezoids that comprise any given pocket correspond to a subtree. Each lid of $VP(i)$ corresponds to an edge in the tree, that separates the subtree of the adjacent pocket from the rest of the tree. For each lid (including $\rho_u(l_i)$), we begin at the corresponding edge in the dual graph, and traverse through the subtree of the pocket, assigning to each node a pointer to the lid. Therefore, if the planar point location query encounters a trapezoid with a pointer to a lid, the lid query returns that lid. We see that the chains $P_1, \ldots, P_h$ can be preprocessed in $O(n)$ total time to handle lid queries in time $O(\log \bar{n})$, where $\bar{n}$ is the number of vertices on the largest of $P_1, \ldots, P_h$.

## 4.1 The Profile for a Set of Stabbed Polygons

Given a collection of polygonal chains $P_1, \ldots, P_h$, all of which are stabbed by a vertical line $\pi$, it is possible to compute the visibility profile $VP(1, \ldots, h)$ of the chains in $O(n + h \log \bar{n} + h \log h)$ time, where $n$ is the total number of vertices of the chains and $\bar{n}$ is the number of vertices on the largest chain. The procedure sorts the chains in terms of the $y$-coordinates of the *bottom points* $b_i$ of the chains, where $b_i$ is the point of $P_i \cap \pi$ of least $y$-coordinate. In this manner, the procedure incrementally adds a profile $VP(k)$ to the current profile $VP(1, \ldots, k-1)$; since $b_k$ is above $b_i$ for $i \in \{1, \ldots, k-1\}$, we are able to show that $VP(k)$ contributes at most one piece to $VP(1, \ldots, k)$. This means that $VP(k)$ can be added efficiently to $VP(1, \ldots, k-1)$ if we can find an $x$-coordinate $x_k$ with the lid property for a point of $P_k$ and the profile $VP(1, \ldots, k-1)$. By maintaining some additional information about $VP(1 \ldots, k-1)$, the procedure finds $x_k$ with a constant number of lid queries, and constructs $VP(1, \ldots, k)$ from $VP(k)$ and $VP(1, \ldots, k-1)$ in $O(n_k + n'_k + \log \bar{n} + \log h)$ time, where $n_k$ is the number of vertices of $P_k$, and $n'_k$ is the number of vertices of $VP(1 \ldots, k-1)$ not on $VP(1, \ldots, k)$. With the initial sorting of the bottom points, this yields the claimed time complexity.

## 4.2 Merging

We describe the merging of the visibility profiles of two subsets $\mathcal{S}'$ and $\mathcal{S}''$ of $\mathcal{S} = \{S_1, \ldots, S_{\lceil \log h \rceil}\}$, where $max\{i | S_i \in \mathcal{S}'\} < min\{j | S_j \in \mathcal{S}''\}$. We have a family of vertical lines such that every chain in $\mathcal{S}'$ is stabbed by at least one of the lines, but no chain of $\mathcal{S}''$ is stabbed by a line. Since the vertical lines separate the elements of $\mathcal{S}''$, we can individually consider the interval between each pair of consecutive lines. Therefore, we consider a vertical strip bordered by the lines $\pi_l$ and $\pi_r$. All chains of $\mathcal{S}'$ that appear in $VP(\mathcal{S}')$ in the strip are stabbed by either $\pi_l$ or $\pi_r$, and no chain of $\mathcal{S}''$ appearing in $VP(\mathcal{S}'')$ is stabbed by either line.

Inductively, we assume that we have, for $VP(\mathcal{S}')$ ($VP(\mathcal{S}'')$) over the strip, a sorted list of all leaps, and for each leap $x$ of $VP(\mathcal{S}')$ ($VP(\mathcal{S}'')$), a pointer to $VP(\mathcal{S}'; x)$ ($VP(\mathcal{S}''; x)$). We merge the lists to form a single sorted list $x_1, \ldots, x_K$ of all leaps in $VP(\mathcal{S}')$ and $VP(\mathcal{S}'')$. For a leap $x_k$ from $VP(\mathcal{S}')$ ($VP(\mathcal{S}'')$), we must compute a pointer to $VP(\mathcal{S}''; x_k)$ ($VP(\mathcal{S}'; x_k)$). We do this through lid queries, as follows. We can assume that in forming the list $x_1, \ldots, x_K$, every leap $x_k$ from $VP(\mathcal{S}')$ knows which profile from $\mathcal{S}''$ contributes $VP(\mathcal{S}''; x_k)$ (this consists of knowing the leaps from $VP(\mathcal{S}'')$ that are nearest to $x_k$ to the left and right). Since we know the profile $VP(j)$ that contributes $VP(\mathcal{S}''; x_k)$, we can compute $VP(j; x_k) = VP(\mathcal{S}''; x_k)$ by querying the point $(x_i, 0)$ on $P_j$. Since $(x_k, 0)$ lies below $VP(j)$ (by our assumption that chains lie above the $x$-axis), the query returns $VP(j; x_k)$.

We define a *subpiece* of $VP(\mathcal{S}')$ or $VP(\mathcal{S}'')$ as the portion of the profile between two consecutive leaps in the merged list $x_1, \ldots, x_K$. Note that a subpiece is a subset of some piece. The following lemma motivates the merge procedure:

**Lemma 5** *Suppose we have a subpiece over the interval $[x_k, x_{k+1}]$, such that $VP(i)$ and $VP(j)$ contribute this subpiece to $VP(\mathcal{S}')$ and $VP(\mathcal{S}'')$, respectively. Then $VP(\mathcal{S}')$ is below $VP(\mathcal{S}'')$ somewhere in $[x_k, x_{k+1}]$ only if $VP(i)$ is below $VP(j)$ at $x_k$ or $x_{k+1}$.*

Now we combine the two profiles over the subpiece $[x_k, x_{k+1}]$. If $VP(j)$ is below $VP(i)$ at both $x_k$ and $x_{k+1}$, then the entire subpiece is contributed by $VP(j)$. Below we describe a procedure for the case when $VP(i)$ is below at one of $x_k$ and $x_{k+1}$, and $VP(j)$ is below at the other. We then show how this procedure can be modified to handle the case where $VP(i)$ is below at both $x_k$ and $x_{k+1}$.

If $VP(i)$ is below at one of $x_k$ and $x_{k+1}$, and $VP(j)$ is below at the other, then our task consists of finding the unique leap in $VP(i,j)$ between $x_k$ and $x_{k+1}$, without traversing portions that are part of $VP(\mathcal{S}' \cup \mathcal{S}'')$. A naive scheme could take time linear in the size of the portions of $VP(\mathcal{S}')$ and $VP(\mathcal{S}'')$ between $x_k$ and $x_{k+1}$, but our approach requires only polylog time. Assume that the vertices of each original profile have been numbered in left-to-right order, and placed in a data structure so that if we are given a pointer to a vertex of the profile, we can in constant time return the numbering of the vertex. Our procedure maintains two pointers to $VP(i)$, denoted $p_l^i$ and $p_r^i$, which are initialized to $VP(i; x_k)$ and $VP(i; x_{k+1})$, and pointers $p_l^j$ and $p_r^j$ to $VP(j)$, initialized to $VP(j; x_k)$ and $VP(j; x_{k+1})$. The pointers $p_l^i$ and $p_l^j$ will be maintained at the same $x$-coordinate, as will $p_r^i$ and $p_r^j$. Initially we know that there is exactly one leap of $VP(i,j)$ between $x(p_l^i) = x(p_l^j)$ and $x(p_r^i) = x(p_r^j)$; the procedure maintains this property while moving $x(p_l^i) = x(p_l^j)$ and $x(p_r^i) = x(p_r^j)$ closer together, eventually sandwiching the leap. The procedure alternates steps on the pointer pairs $(p_l^i, p_r^i)$ and $(p_l^j, p_r^j)$. We describe a step on the pair $(p_l^i, p_r^i)$:

1. Query the numbering of the vertices of $VP(i)$ nearest $p_l^i$ and $p_r^i$, and assign these numberings to $p_l^i$ and $p_r^i$.

2. Find $q$, the vertex of $VP(i)$ whose numbering is halfway between the numberings of $p_l^i$ and $p_r^i$.

3. Compute $VP(j; x(q))$.

4. Compare the $y$-coordinates of $VP(j; x(q))$ and $q = VP(i; x(q))$; this tells us whether the leap is left or right of $x(q)$; accordingly, update either $p_l^i$ and $p_l^j$, or $p_r^i$ and $p_r^j$.

Upon completion of this step on the pair $(p_l^i, p_r^i)$, perform a symmetric step on $(p_l^j, p_r^j)$, and continue to alternate the steps. Eventually the total number of vertices on $VP(i)$ between $p_l^i$ and $p_r^i$ and on $VP(j)$ between $p_l^j$ and $p_r^j$ is less than a small, pre-set constant, so in constant time we find the leap and complete this subpiece of $VP(\mathcal{S}' \cup \mathcal{S}'')$.

A slight modification of the above procedure handles the case where $VP(i)$ is below $VP(j)$ at both $x_k$ and $x_{k+1}$. Query $VP(j; x_k)$ on $VP(i)$, to find a lid $\overline{ab}$ which $VP(j)$ must cross in order to contribute a piece to $VP(i,j)$. If $x(a) \notin [x_k, x_{k+1}]$, then $VP(j)$ is not below $VP(i)$ anywhere in the interval $[x_k, x_{k+1}]$. If $x(a) \in [x_k, x_{k+1}]$, then perform a lid query on the point $(x(a), 0)$ to compute $VP(j; x(a))$. If $VP(i)$ is below $VP(j)$ at $x(a)$, then

the subpiece between $x_k$ and $x_{k+1}$ is contributed totally by $VP(i)$; otherwise, we break the subpiece $[x_k, x_{k+1}]$ into two subpieces, $[x_k, x(a)]$ and $[x(a), x_{k+1}]$, and process each subpiece with the above procedure.

Consider the total time of merging $VP(\mathcal{S}')$ and $VP(\mathcal{S}'')$. Let $\bar{h}$ represent the total number of chains in $\mathcal{S}'$ and $\mathcal{S}''$, and $\bar{n}$ the number of vertices on the largest chain in the set $\mathcal{S}' \cup \mathcal{S}''$. Creating the combined sorted lists of leaps $x_1, \ldots, x_K$ for all strips takes time $O(\bar{h})$, because we already have the sorted lists of leaps for $\mathcal{S}'$ and $\mathcal{S}''$ separately. Computing $VP(i; x_k)$ and $VP(j; x_k)$ for every leap $x_k$ requires time $O(\bar{h} \log \bar{n})$, since it consists of performing one lid query per leap. We then process each of the $O(\bar{h})$ subpieces separately, perhaps breaking some subpieces into two subpieces with the help of a single lid query. Processing a subpiece consists of alternating steps on the pairs of pointers $(p_l^i, p_r^i)$ and $(p_l^j, p_r^j)$. Each step consists of one lid query plus some constant time work, and is therefore $O(\log \bar{n})$. Because every two steps eliminate at least half of the vertices of $VP(i)$ between $p_l^i$ and $p_r^i$ and of $VP(j)$ between $p_l^j$ and $p_r^j$, the number of steps is $O(\log \bar{n})$. Therefore each subpiece requires $O(\log^2 \bar{n})$ time, for a total of $O(\bar{h} \log^2 \bar{n})$ time to merge $VP(\mathcal{S}')$ and $VP(\mathcal{S}'')$.

## 4.3 Putting It Together: An Optimal Algorithm

The above subsections describe how to compute $VP(\mathcal{P})$ for $\mathcal{P} = \{P_1, \ldots, P_h\}$ in time $O(n + h \log \log h \log^2 \bar{n})$, where $n$ is the total number of vertices in $\mathcal{P}$, and $\bar{n}$ is the number of vertices on the largest chain of $\mathcal{P}$. A modification allows this algorithm to compute $VP(\mathcal{P})$ in optimal $\Theta(n + h \log h)$ time. The modification consists of breaking $\mathcal{P}$ into two groups, the "large" chains and the "small" ones, computing the visibility profile of each group separately, and then merging the profiles with a final linear-time merge.

The first observation to be made is that if $h = O(n/\log^3 n)$, then the algorithm's complexity is $O(n)$. Motivated by this observation, we break $\mathcal{P}$ into two groups as follows: all chains in $\mathcal{P}$ with more than $\log^3 n$ vertices are placed in the "large" group, and the rest in the "small" group. The large group can have no more than $n/\log^3 n$ members, so the algorithm can compute the visibility profile of this group in $O(n)$ time.

Assume that the small group has $h = \Omega(n/\log^3 n)$ members (if not, the algorithm is $O(n)$ on this group). No chain of the small group has more than $\log^3 n$ vertices, implying that $\bar{n} \leq \log^3 n$. The complexity of the algorithm is therefore $O(n + h \log \log h \log^2(\log^3 n)) = O(n + h \log \log h (\log \log n)^2)$. Since $h = \Omega(n/\log^3 n)$, we have that $\log \log n = O(\log \log h)$, giving a complexity of $O(n + h(\log \log h)^3) = O(n + h \log h)$. Therefore, the visibility profiles of both the small group and the large group can be computed in $O(n + h \log h)$, giving $VP(\mathcal{P})$ in the same time bound.

# References

[AM] E. Arkin and J.S.B. Mitchell, "An Optimal Visibility Algorithm for a Simple Polygon With Star-Shaped Holes", Technical Report No. 746 School of Operations Research and Industrial

Engineering, Cornell University, June, 1987.

[AAGHI] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai, "Visibility of Disjoint Polygons", *Algorithmica*, Vol. 1, (1986), pp. 49-63.

[Ch1] B. Chazelle, "Efficient Polygon Triangulation", CS-TR-249-90, Princeton Univ., 1990.

[Ch2] B. Chazelle, "Triangulating a Simple Polygon in Linear Time", CS-TR-264-90, Princeton Univ., May 1990.

[CTV] K. Clarkson, R.E. Tarjan, C. Van Wyk, "A Fast Las Vegas Algorithm for Triangulating a Simple Polygon", *Proc. Fourth Annual ACM Symposium on Computational Geometry*, pp. 18-22, 1988. Also Princeton Technical Report CS-TR-157-88.

[EA] H.A. El Gindy and D. Avis, "A Linear Algorithm for Computing the Visibility Polygon From a Point", *Journal of Algorithms*, Vol. 2 (1981), pp. 186-197.

[JS] B. Joe and R.B. Simpson, "Correction to Lee's Visibility Polygon Algorithm", *BIT*, **27** (1987), pp. 458-473.

[Ki] D.G. Kirkpatrick, "Optimal Search in Planar Subdivisions", *SIAM Journal on Computing*, **12** (1983), No. 1, pp. 28-35.

[KKT] D.G. Kirkpatrick, M.M. Klawe, and R.E. Tarjan, "Polygon Triangulation in $O(n \log\log n)$ Time with Simple Data-Structures", *Proc. Sixth Annual ACM Symposium on Computational Geometry*, Berkeley, CA, June 6-8, 1990, pp. 34-43.

[Le1] D.T. Lee, "Proximity and Reachability in the Plane", Ph.D. Thesis, Report R-831, Dept. of Electrical Engineering, University of Illinois at Urbana-Champaign, Nov. 1978.

[Le2] D.T. Lee, "Visibility of a Simple Polygon", *Computer Vision, Graphics, and Image Processing*, Vol. 22 (1983), pp. 207-221.

[O'R] J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford University Press, 1987.

[Se] R. Seidel, "A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons", Manuscript, October, 1990.

[Sh] M. Sharir, "Davenport-Schinzel Sequences and their Geometric Applications", pp 253-278, NATO ASI Series, Vol. F40, Theoretical Foundations of Computer Graphics and CAD, R.A. Earnshaw (Ed.), Springer-Verlag Berlin Heidelberg, 1988.

[SO] S. Suri and J. O'Rourke, "Worst-Case Optimal Algorithms For Constructing Visibility Polygons With Holes", *Proc. Second Annual ACM Symposium on Computational Geometry*, Yorktown Heights, NY, June 1986, pp. 14-23.

[TV] R.E. Tarjan and C. Van Wyk, "An $O(n \log\log n)$-Time Algorithm for Triangulating a Simple Polygon", *SIAM Journal on Computing*, **17** (1988), No. 1, pp. 143-178.

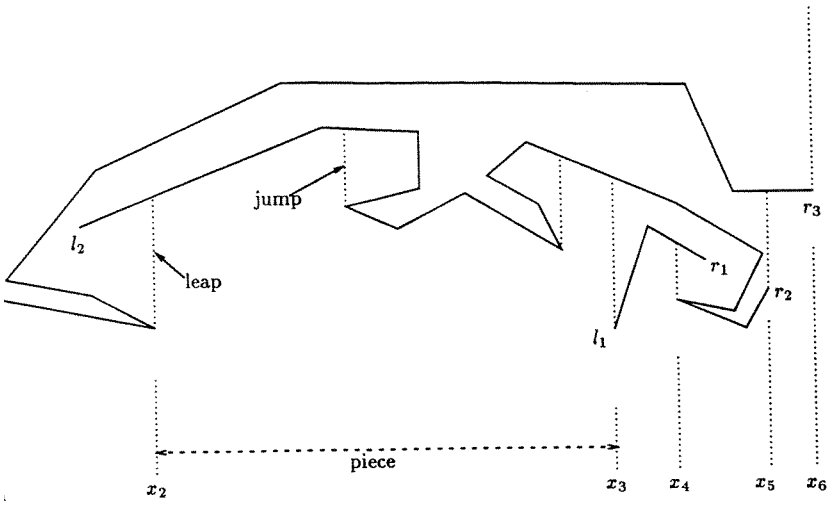


Figure 1: Definition of jumps, leaps, and pieces

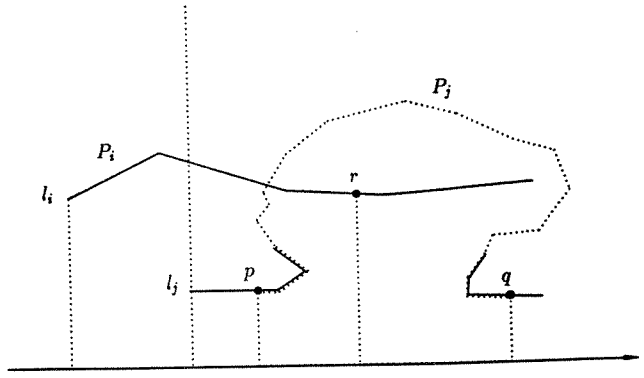


Figure 2: Proof of Lemma 1

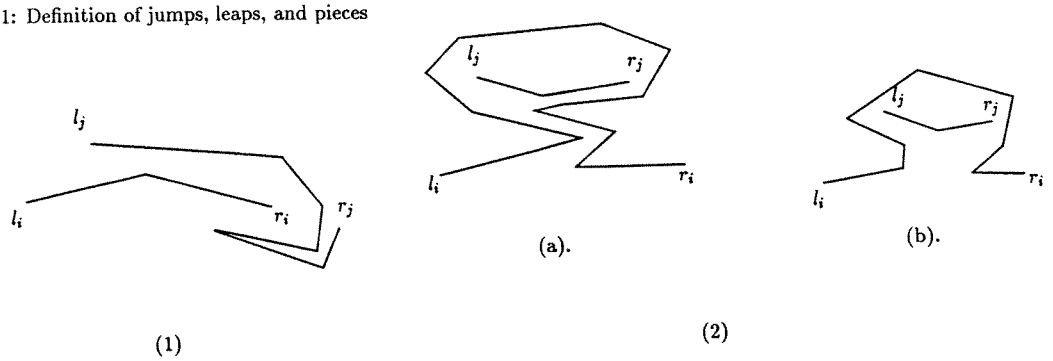


Figure 3: Structure of $VP(i, j)$

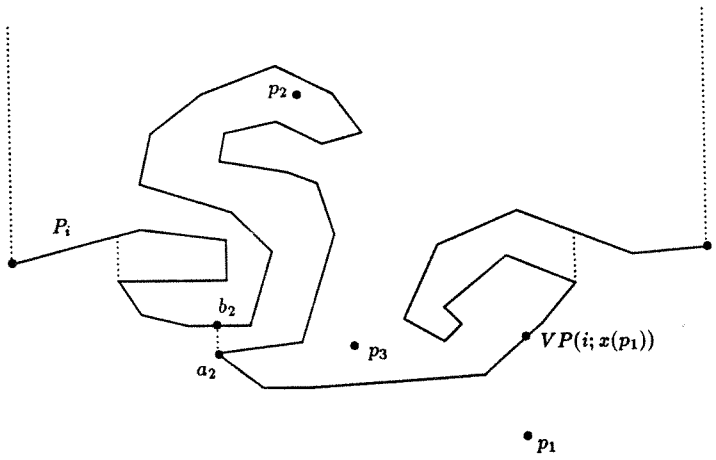


Figure 4: Results of lid queries of points $p_1, p_2, p_3$ on chain $P_i$