SIAM J. COMPUT.
Vol. 20, No. 5, pp. 888–910, October 1991

006

AN OUTPUT-SENSITIVE ALGORITHM FOR COMPUTING VISIBILITY GRAPHS*

SUBIR KUMAR GHOSH† AND DAVID M. MOUNT‡

Abstract. The visibility graph of a set of nonintersecting polygonal obstacles in the plane is an undirected graph whose vertex set consists of the vertices of the obstacles and whose edges are pairs of vertices (u, v) such that the open line segment between u and v does not intersect any of the obstacles. The visibility graph is an important combinatorial structure in computational geometry and is used in applications such as solving visibility problems and computing shortest paths. This paper presents an algorithm that computes the visibility graph of a set of obstacles in time $O(E + n \log n)$, where E is the number of edges in the visibility graph and n is the total number of vertices in all the obstacles.

Key words. visibility graph, output-sensitive algorithms, shortest paths

AMS(MOS) subject classifications. 68Q25, 68U05

1. Introduction.

The *visibility graph* of a set of nonintersecting polygonal obstacles in the plane is a graph whose vertex set consists of the vertices of the obstacles and whose edges are the pairs of vertices (u, v) such that the open line segment between u and v does not intersect any of the obstacles. In this paper an output-sensitive algorithm is presented for computing the visibility graph of a set of polygonal obstacles. The visibility graph is a fundamental combinatorial structure in computational geometry; it is used, for example, in applications such as computing shortest paths amidst polygonal obstacles in the plane [11]. In particular, given a set of polygonal obstacles in the plane, the shortest-length path between any two points s and t travels along the edges of the visibility graph of the obstacle set augmented with the points s and t [10], [15].

In the worst case the visibility graph of a set of obstacles with n total vertices may contain $O(n^2)$ edges. An $O(n^2 \log n)$ algorithm for this problem was given by Lee [9] and Sharir and Schorr [15]. Later, worst case optimal $O(n^2)$ algorithms were discovered by Asano et al. [1] and Welzl [16]. If the visibility graph contains relatively few edges, for example, when there are many densely packed objects, it is desirable to have an algorithm whose running time is a function of the number of edges. Hershberger has described an output-sensitive algorithm for the case of computing the visibility graph within a simple polygon (once the polygon has been triangulated) [6], and Overmars and Welzl have given an algorithm for computing the visibility graph for a set of disjoint polygonal obstacles whose running time is $O(E \log n)$ and whose space is $O(n)$, where E is the number of edges in the visibility graph [14].

In this paper we present an algorithm that computes the visibility graph of an arbitrary set of disjoint obstacles with running time $O(E + n \log n)$. The $O(n \log n)$ term is overhead needed for computing a particular triangulation of the obstacle-free

* Received by the editors July 19, 1989; accepted for publication (in revised form) December 11, 1990. A preliminary version of this paper appeared in the Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 11–19.

† Computer Science Group, Tata Institute of Fundamental Research, Bombay, India. The work of this author was supported by Air Force Office of Scientific Research grant AFOSR-86-0092, while he was visiting the Center for Automation Research at the University of Maryland, College Park, Maryland.

‡ Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742. The work of this author was supported by National Science Foundation grant CCR-8908901.

space, after which the algorithm runs in $O(E)$ time. This is optimal in the worst case with respect to E and n, since there are cases of n obstacles where $E = O(n)$, but computing the visibility graph is equivalent to sorting a set of n points [1]. The algorithm uses $O(E+n)$ space. Recalling the application of computing shortest paths amidst polygonal obstacles, once the visibility graph has been constructed, the shortest path can then be computed in $O(E+n \log n)$ time by Fredman and Tarjan's variation of Dijkstra's algorithm using Fibonacci heaps [2].

The key to the algorithm's efficiency is a way of structuring the edges of the visibility graph in terms of a set of objects called *funnel sequences.* Intuitively, the funnel sequence associated with an edge of an obstacle encodes the set of vertices that can see some portion of this edge. We present a novel technique of traversing the funnel sequences.

Throughout, P will denote a bounded polygonal domain, which we will think of as a simple polygon (forming the *external boundary*) whose interior contains a set of simple polygons (the *holes*) such that the holes have pairwise disjoint interiors. Define the *free space* to be the closed space lying on or within the external boundary and on or outside the holes. (If no exterior boundary is given, the convex hull of the obstacle set, which is computable in $O(n \log n)$ time, suffices as the external boundary.) Throughout, in using the term *polygonal domain,* we will assume the existence of an external boundary. The *boundary* of a polygonal domain is represented by a single counterclockwise cycle of directed edges forming the holes. Thus for each directed edge, free space lies to the left side of the edge. To simplify the presentation, we will make the "general position" assumptions throughout that no three vertices of the polygonal domain are collinear and no two vertices share the same x-coordinates. We will also assume that each vertex of P is incident on exactly two edges of P (line segment obstacles can be handled as degenerate polygons with two oppositely directed edges). Our results hold in the absence of these assumptions, but the presentation would be complicated by a number of tedious special cases that would need to be considered.

2. The plane-sweep triangulation. As mentioned in the Introduction, the visibility graph algorithm is based on a triangulation of free space. Let $T_1, T_2, \cdots, T_m$ denote the triangles of this triangulation. Thus the free space region defined by P is just the union of these triangles: $P = \bigcup_{i=1}^{m} T_i$. The visibility graph algorithm operates by constructing a series of subsets of free space by successively adjoining triangles to one another, $P_1 = T_1$, $P_2 = T_1 \cup T_2$, $P_3 = T_1 \cup T_2 \cup T_3$, etc. We compute a complete visibility graph for each subset P_k by augmenting the visibility graph for P_{k-1}. (To be exact, we simultaneously add a number of triangles incident on a single vertex.) Because of the nature of the augmentation procedure, it will be important to select the triangulation and the ordering of triangles in a careful way. Fortunately, there is a simple and natural triangulation based on plane-sweep which suffices for our purposes. The triangulation algorithm is essentially equivalent to one described by Mehlhorn [13, pp. 160–172]. Although Mehlhorn's algorithm assumes that the polygon has no holes, the algorithm generalizes easily.

The idea behind the plane-sweep triangulation for polygons is most easily illustrated by describing the plane-sweep triangulation of a set of points $p_1, p_2, \cdots, p_n$. As is common in plane-sweep algorithms, first the points are sorted in increasing order of their x-coordinates. The triangulation initially contains no edges, just the vertex whose x-coordinate is minimum. Inductively, let us assume that the first $k-1$ points have been triangulated. Think of the outer boundary of the triangulated region as a

polygon P_{k-1}, namely the convex hull of the first $k-1$ points. Clearly the point p_k lies outside of P_{k-1}. Thus we can incorporate p_k into the triangulation by connecting p_k to all of the points on the boundary of P_{k-1} that are visible from p_k (thinking of P_{k-1} as an obstacle). The point p_k will be joined to an inward-convex chain of vertices on the boundary of P_{k-1}.

The plane-sweep triangulation of the interior of a polygonal domain is similar. First the vertices are sorted by x-coordinate. Let $v_1, v_2, \cdots, v_n$ denote the resulting sequence. Inductively assume that the first $k-1$ vertices have been incorporated into the triangulation. The outer boundary of the triangulated region consists of a set of disjoint simple polygons, which may degenerate to isolated points and line segments. Thinking of the edges of the polygonal domain as forming obstacles, the vertex v_k is incorporated into the triangulation by adding visible segments between v_k and all its visible neighbors on the boundary of the triangulated region.

The plane-sweep triangulation can be built in $O(n \log n)$ time. The important property of the plane-sweep triangulation, which will be exploited by our algorithm, is summarized in the next lemma. This lemma follows from the discussion in [13]. See Fig. 1.

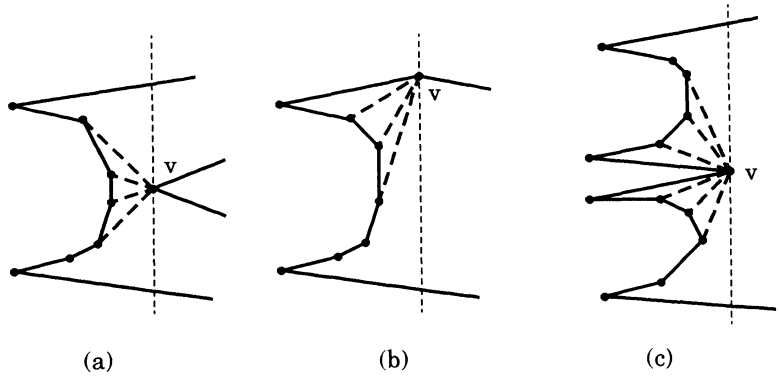

FIG. 1. *Connecting a point to the triangulation.*

LEMMA 2.1 (Mehlhorn [13]). *Consider the triangles formed as an arbitrary vertex v is incorporated into the triangulation of a polygonal domain P. These triangles form either one or two connected sequences about v such that the sides opposite v form an inward-convex chain with respect to v (degenerating possibly to a single point). If there are two such sequences, then these sequences are separated from one another by the boundary of P (Fig. 1(c)).*

3. The funnel sequence. The visibility graph of a polygonal domain possesses a great deal of structure when seen within the context of the polygon itself. In this section we describe the fundamental structure that our algorithm manipulates, called the *funnel sequence* for an edge of the polygonal domain P. Funnels arise naturally in shortest-path and visibility problems in simple polygons [5], [6], [10]. We begin with some definitions and observations about funnels.

Define a *visible chain* in polygonal domain P to be a path in the visibility graph of P. To avoid confusion, we will use the term *edge* when referring to an edge of a polygon, and the term *visible segment* or just *segment* when referring to an edge in a visibility graph. A chain is *convex* if the figure defined by joining the two endpoints

of the chain is a convex body. Consider a vertex v that is visible from an interior point z of an edge (x, y) of P. For the sake of illustration, imagine that the edge (x, y) is directed upwards and point v is to the left of the edge (see Fig. 2). Define the *lower chain* of v with respect to (x, y) to be the unique convex visible chain from v to x such that the interior region bounded by this chain and by the line segments vz and xz is empty. Intuitively, the lower chain is formed by imagining that the segment vz is a rubber band and sliding the point z of this rubber band down the edge (x, y) until reaching x. The upper chain of v with respect to (x, y) is defined analogously for y. The lower chain, upper chain, and edge (x, y) bound a simple polygon in P which we call a *funnel.*

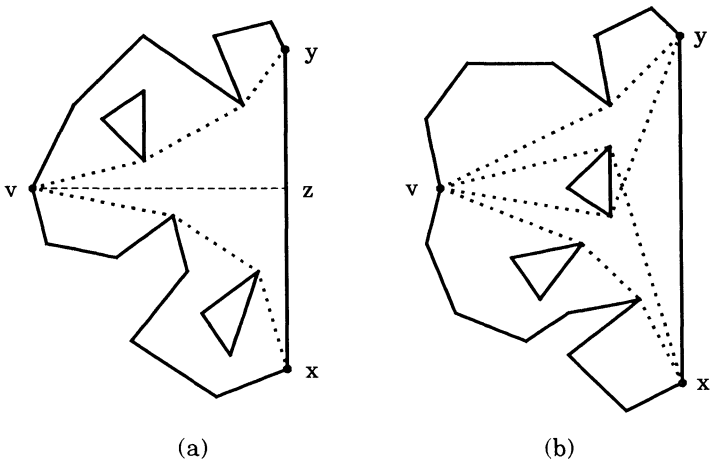

FIG. 2. *Visible chains and funnels.*

By definition, the interior of the funnel contains no vertices and no edges of P. The vertex v is called the *apex* of the funnel and the edge (x, y) is the *base* of the funnel. Unlike funnels that arise in simple polygons [6], in polygonal domains there may be many funnels sharing a single apex vertex (see Fig. 2(b)). We will think of these apexes as being distinct objects occupying the same physical location in space.

Considering the visibility graph for P and a vertex v of P. Let $u_0, u_2, \cdots, u_m$ be the clockwise sequence of vertices that are visible from v so that (u_0, v) and (v, u_m) are edges of P. For every pair of cyclically adjacent vertices u_{i-1} and u_i, there is a unique edge e of the polygonal domain that can be seen by an observer located at v looking between these vertices (for otherwise, there would be another visible vertex between them). Thus there is a unique funnel whose apex is v, whose base is e, whose upper chain begins with (v, u_{i-1}), and whose lower chain begins with (v, u_i). Given the first directed segment (v, u_i) of the lower chain, the first directed segment (v, u_{i-1}) of the upper chain is uniquely determined, and vice versa. An immediate result of this correspondence is the following.

LEMMA 3.1. *There is a* 1-1 *correspondence between pairs of cyclically adjacent directed segments of the visibility graph about a vertex* v, $((v, u_{i-1}), (v, u_i))$ *for* $0 < i \leqq m$, *and the funnels whose apex is* v.

COROLLARY. *The total number of funnels in a visibility graph with* E *undirected edges and* n *vertices is* $2(E - n)$, *which is* $O(E)$.

For a given edge (x, y) of the polygonal domain P, let FNL(x, y) denote the set of funnels whose base edge is (x, y). Recall that the interior of P lies to the left of the edge, so these funnels all lie on the left of (x, y). For completeness, vertices x and y can each be thought of as the apexes of degenerate funnels in FNL(x, y). (There are $2n$ degenerate funnels, so this does not alter the number of funnels asymptotically.) If v is the apex of a funnel in FNL(x, y), and u is the first vertex on the lower chain from v to x, then u is visible from the edge (x, y) implying (by convexity of funnels) that u is the apex of a unique funnel that is contained within v's funnel. If we think of the apex u as the parent of the apex v, we see that the set of funnels in FNL(x, y) forms a tree rooted at x whose paths are the lower chains of FNL (x, y). Note that it is important to distinguish vertices from apexes here because the same vertex can appear many times as an apex in FNL (x, y), whereas each apex can appear only once. Each path from a node to the root of this tree is a convex visible chain that turns clockwise. Call this the *lower tree* for the edge (x, y) (see Fig. 3(a)). Analogously, we define the *upper tree* to consist of the tree of upper chains of FNL (x, y) rooted at y (see Fig. 3(b)). Paths from a node to the root in the upper tree are convex visible chains that turn counterclockwise.

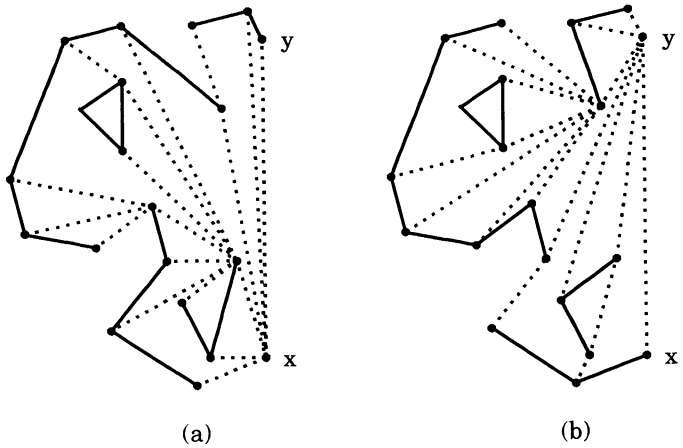

FIG. 3. *The lower and upper trees.*

We can define a natural linear ordering on the funnels of FNL (x, y) based on these trees by considering the clockwise preorder traversal of the lower tree. There is another natural order that results from considering a clockwise postorder traversal of the upper tree. In both orderings, the degenerate funnel at x is first, and the degenerate funnel at y is last. Our next result states that these orders are in fact equal to one another. We refer to this clockwise ordering of funnels as the *funnel sequence* for the edge (x, y).

LEMMA 3.2. *The linear orders on* FNL (x, y) *arising from a clockwise preorder traversal of the lower tree and a clockwise postorder traversal of the upper tree are the same.*

Proof. Let f_1 and f_2 be two funnels so that f_1 precedes f_2 in a clockwise preorder traversal of the lower tree. Think of the lower chains of f_1 and f_2 as paths from the root x to the apexes of these chains, and think of upper chains as paths from y. There are two reasons that f_1 may precede f_2: (1) the lower chain of f_1 is a subchain of the lower chain of f_2 and (2) the lower chain of f_2 diverges clockwise from f_1's lower chain at some common ancestor.

In case (1) the apex of f_1 lies on the lower chain of f_2. By the emptiness of funnel f_2, the upper chain of f_1 contains a single segment that lies entirely within f_2 and joins the apex of f_1 to a vertex v on the upper chain of f_2 (either at a point of tangency or at y). See Fig. 4(a). This implies that the upper chain for f_2 diverges clockwise from the upper chain for f_1 at v, and hence f_1 precedes f_2 in any clockwise traversal of the upper tree.

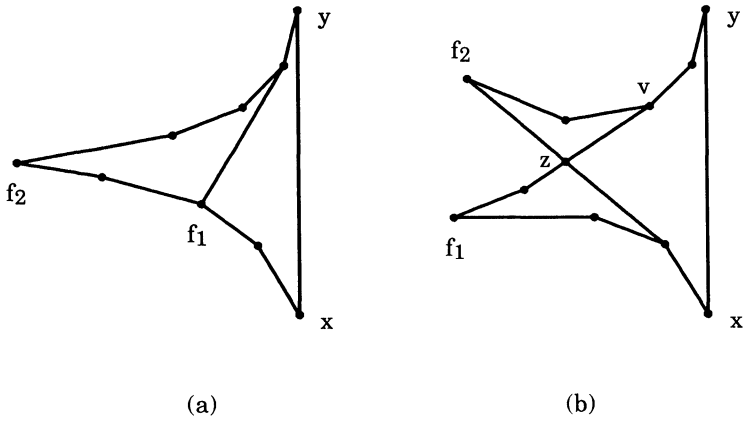

FIG. 4. *Funnel ordering.*

In case (2) f_2's path diverges from f_1 along a segment that enters into the interior of f_1. Since funnels are empty, this segment must eventually intersect the boundary of f_1 at some point z (which may or may not be a vertex). The point z must lie on the upper chain of f_1, since it cannot intersect the interior of edge (x, y), and by convexity it cannot cross the lower chain of f_1. See Fig. 4(b). If z is the apex of f_2, implying that f_2 is an ancestor of f_1 in the upper tree, then f_1 precedes f_2 in any postorder traversal of the upper tree. Otherwise the lower chain of f_2 crosses the upper chain of f_1 at z. The upper chain of f_2 cannot enter into the interior of f_1 by the emptiness of funnels, and hence the upper chain of f_2 must diverge clockwise from the upper chain f_1 at some vertex v before reaching z. This implies that the upper chain for f_1 precedes f_2 in any clockwise traversal of the upper tree. □

4. The enhanced visibility graph. In this section we describe the basics of the visibility graph algorithm. We assume that we have computed the plane-sweep triangulation for the polygonal domain P. (Actually, the process described here could be performed while the triangulation is being built.) Recall from § 2 that the vertices $v_1, v_2, \cdots, v_n$ of P have been sorted in increasing order by x-coordinate, and they are incorporated into the triangulation in this order. Let P_k denote the triangulated region containing the vertices $v_1, \cdots, v_k$. We will think of P_k as a polygonal domain contained within P (it may be disconnected and contain isolated points and edges).

For each k we maintain a structure called the *enhanced visibility graph* for P_k. Before specifying the enhanced visibility graph we first give some definitions. Consider a vertex v in the visibility graph and consider the visible segments directed out of v. This list will include the two boundary edges of P incident on v. Let (v, u) be a visible segment incident on v. Define the *clockwise successor* of (v, u), CW (v, u), to be the next visible segment about v in clockwise order and define the *counterclockwise successor* of (v, u), CCW (v, u), analogously. Define the *clockwise extension* CX (u, v) of a visible

segment directed into v as follows (note the reversal of arguments). Rotate the ray from v through u clockwise by 180 degrees about v. If this sweep lies entirely within the interior of P locally about v, then the extension is the very next visible segment encountered after the 180 degree sweep (by our assumption of the noncollinearity of three vertices, there will be no segment at exactly 180 degrees). If not, then the clockwise extension is undefined. The *counterclockwise extension*, CCX (u, v), is defined symmetrically using a counterclockwise sweep. Fig. 5 illustrates three of these entities, and the fourth, CX (u, v), is undefined for this example. Finally, define REV (u, v) to be the directed reversal (v, u).

DEFINITION. Define the *enhanced visibility graph* for polygon P to consist of:

- the boundary of P represented such that the two neighbors of a given vertex can be found in constant time;

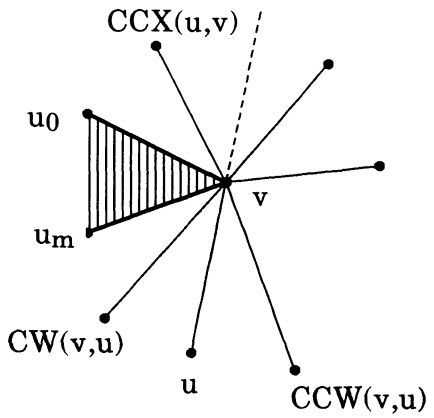

FIG. 5. *Traversal primitives.*

- the visibility graph for P, represented such that the operations CCW, CW, CCX, CX, and REV can be evaluated in constant time each; and
- the funnel sequence FNL (x, y), for each edge (x, y) on the boundary of P, represented (say, as a doubly linked list) so that the operations of split, concatenate, predecessor, and successor can be performed in constant time each. (To be exact, our algorithm only maintains the funnel sequence for a selected set of boundary edges along the right side of P, along which we will augment the triangulation.)

In § 7 we will show how to implement CW, CCW, CX, CCX, and REV, but for now we assume that these operations are available to us. From Lemma 3.1 we may assume that each funnel apex is uniquely represented by giving the first segment in its lower chain (directed out of the funnel's apex), but we will refer to apexes by vertices, when the funnel is clear from context. Next we observe that the enhanced representation of the visibility graph contains sufficient information to permit traversals of the upper and lower trees.

LEMMA 4.1. *Consider the enhanced visibility graph of a polygonal domain P, and suppose that (u, v) is any directed segment of the lower tree of an edge (x, y) of P, such that u is a parent of v. The following relatives of u and v in the lower tree can be computed in constant time*:

(i) *the parent of u,*
(ii) *the extreme clockwise and counterclockwise children of v, and*
(iii) *the clockwise and counterclockwise siblings of v.*

Analogous claims hold for the upper tree.

Proof. We prove the lemma for lower trees, and a symmetric argument establishes the result for upper trees. For (i), by the clockwise turning of the lower chains, the parent of u in the lower tree is the apex whose lower chain begins with the head vertex of the clockwise extension CX (v, u), that is, CX (REV (u, v)), provided it exists (see Fig. 6). If this extension is undefined, then it follows (by the emptiness of funnels) that $u = x$, and hence u is the root of the tree.

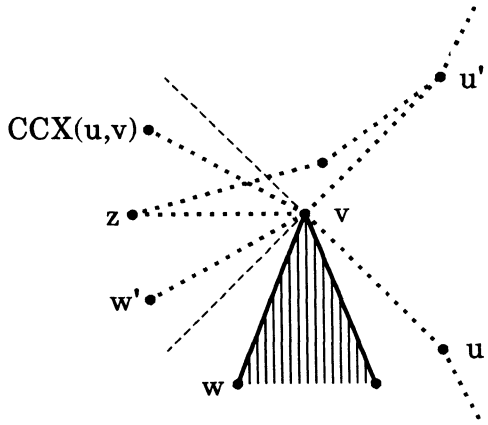

FIG. 6. *Tree traversal.*

Since (u, v) is a segment of the lower tree for (x, y), v is the apex of a funnel that is visible from the interior of the edge (x, y). The first lower chain segment of this funnel is (v, u) and the first upper chain segment is CCW (v, u). Let $(v, u') = \text{CCW}(v, u)$. To establish (ii), let (w, v) be the directed edge on the polygon's boundary, such that the interior of the polygon lies to the left of this directed edge. If the counterclockwise extension CCX (u, v) is undefined (implying that w lies in the halfplane to the right of segment (u, v)), then, by the convexity of lower chains, v cannot have a child in the lower tree, and hence is a leaf. Otherwise, any children of v must lie between CCX (u, v) and (v, w) counterclockwise about v (see Fig. 6). Let z be such a vertex. For z to be a child of v, the funnel with apex z whose lower chain begins with segment (z, v) must be visible from the interior of edge (x, y). This is true if and only if the counterclockwise angle $u'vz$ is less than 180 degrees. (The "only if" part of this statement is true from the convexity of the upper chains. The "if" part holds because if $u'vz$ is less than 180 degrees, then the ray from z through v passes through the interior of the funnel whose apex is v and strikes the interior of the edge (x, y).)

Let w' be chosen such that if the counterclockwise extension CCX (u', v) exists, then $(v, w') = \text{CW}(\text{CCX}(u', v))$, and otherwise $w' = w$. Clearly, w' is computable in constant time, and it follows from the previous discussion that the children of v are exactly those apexes z visible from v that lie counterclockwise from the head of CCX (u, v) to w', assuming that this angular sector is not empty. If so, the reversal of these two edges, REV(CCX (u, v)) and REV (v, w'), are the first edges of the lower chains of the extreme clockwise and counterclockwise children of v, respectively. If the sector is empty, then v is a leaf.

For (iii), note that the clockwise sibling of v is just CW (u, v), provided that v is not the extreme clockwise child of u. A symmetric statement holds for the counterclockwise sibling of v. By (ii) we can test whether v is an extreme child of u. □

COROLLARY. *Given the enhanced visibility graph, clockwise and counterclockwise traversals of the lower and upper trees can be performed in time proportional to the sizes of the trees, and a funnel can be traversed in time proportional to its size.*

5. Splitting the funnel sequence. Our next task is to describe how to use the ability to traverse the enhanced visibility graph in order to add a new vertex into the visibility graph.

The basic loop of the visibility graph algorithm consists of successively adding triangles from the triangulation of the polygonal domain and updating the visibility graph with each addition. We assume inductively that an enhanced visibility graph has been computed for the interior of the triangulated region so far. For each new triangle added, we update the visibility graph appropriately. The fundamental operation on which our algorithm is based is procedure SPLIT. This procedure is given an enhanced visibility graph for a polygonal domain P, a directed edge (x, y) on the external boundary of P, and a point v lying to the right of this edge so that the triangle xvy is external to P. The procedure essentially merges the triangle xvy into P (erasing the edge (x, y)) and computed the enhanced visibility graph of the resulting polygonal domain.

After the edge (x, y) is removed, every vertex in P that was visible from some interior point of the edge (x, y) will be visible from either the interior of edge (x, v) or edge (v, y) or both. The apex of a funnel of P is visible from both edges (x, v) and (v, y) (through the funnel) if and only if the apex is visible from v. Consider a funnel with apex u, whose upper chain is U and whose lower chain is L. (We will often refer to a funnel by giving the name of the vertex that is its apex whenever the actual funnel is clear from context.) If u can see v through the funnel, then SPLIT will add the visible segment between u and v, in effect splitting the funnel u into two funnels, one for FNL (x, v) whose lower chain consists of L and whose upper chain has only the segment (u, v), and one for FNL (v, y) whose upper chain consists of U and whose lower chain has only the segment (u, v) (see Fig. 7(a)). If u can see only one edge through the funnel, say the lower edge (x, v), then SPLIT will make u the apex of a funnel to be added to FNL (x, v). The lower chain of such a funnel will consist of L, and the upper chain will consist of a tangent segment from v to the upper chain U, followed by the remainder of U to u (see Fig. 7(b)).

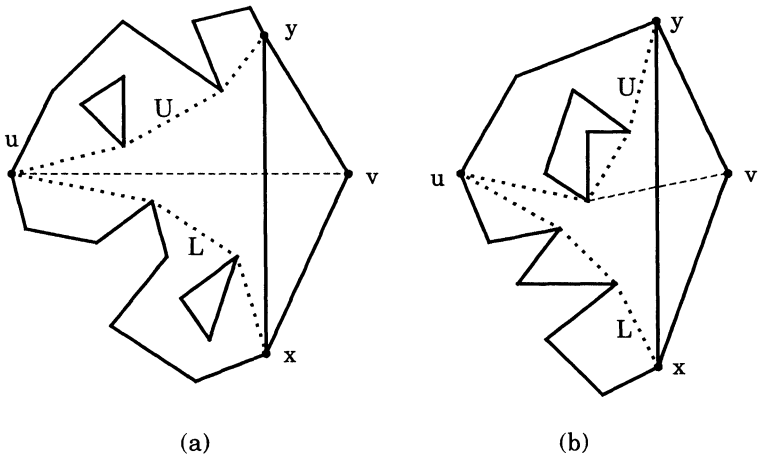

FIG. 7. *Splitting a funnel.*

To illustrate the operation of SPLIT in greater detail, consider two funnel apexes u and t that are visible from v in consecutive clockwise order about v. Between these apexes lies a *pocket* of visibility, where there may exist apexes that can see the edge (x, y), but not the vertex v. Extend the visible segments (v, u) and (v, t) until reaching points q' and r' on the boundary of P (see Fig. 8). There are no vertices or polygonal edges in the triangle $vq'r'$ because there are no visible vertices between u and t. Imagine for the moment two funnels whose apexes are the points q' and r'. These points are visible from the interior of edge (x, y), and hence (as apexes of two funnels that pass between u and t) they can be put into the linear order of FNL (x, y). It is not hard to see that q' and r' will be consecutive in funnel order and (as will be proved in Lemma 5.2) $q' < r'$. (We will use the notation $<$ and $>$ to relate apexes in funnel order.) Let q be the true apex in FNL (x, y) that precedes q' in funnel order, and let r be the true apex in FNL (x, y) that succeeds r'. It may be that $q = u$ or $r = t$. Intuitively, if $q \neq u$, then every apex a, $u < a \leqq q$, has its visibility of v blocked from below by u. (We say an apex q's visibility of v is *blocked from below* by u if the lower chain of q passes through u, and u is a point of tangency with respect to v on this chain.) These apexes are only visible from the upper edge (v, y). Similarly, if $r \neq t$, then every apex a, $r \leqq a < t$, has its visibility of v blocked from above by t. These apexes are only visible from the lower edge (x, v).

The procedure SPLIT operates by finding the funnel apexes that are visible from v in clockwise order. For each consecutive pair of visible apexes that it finds (such as u and t) there is a pocket of edge visible apexes. The procedure locates apexes (such as q and r) at which the pocket can be split. Since the funnel sequence is a simple doubly linked list, the splitting can be done in constant time, once the endpoints of the split are known. The key to the efficiency of the procedure is to locate t, q, and r quickly, once u is known. The heart of the SPLIT procedure is a search of the enhanced visibility graph, which when given a visible vertex u, finds these entities and, in general, a number of other visible vertices in time proportional to the number of visible pairs encountered. Thus the effort of the algorithm will be amortized against the number of newly discovered visible segments.

Before describing the SPLIT procedure, we investigate the deeper structure of the upper and lower trees. The fundamental intuition that we exploit is that within a

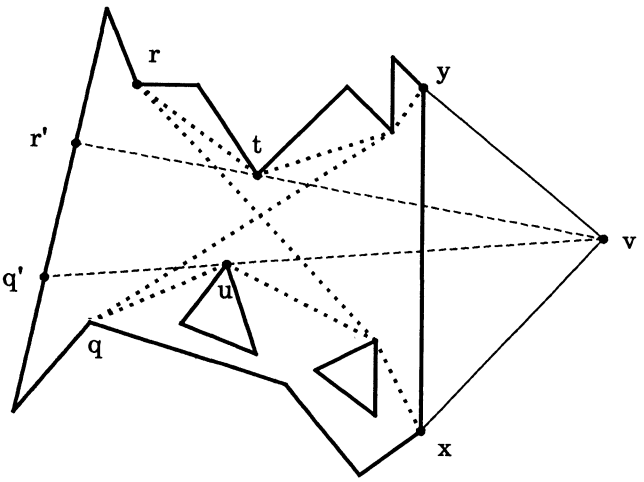

FIG. 8. *Splitting the funnel sequence.*

sufficiently small region, namely two vertices that are adjacent in funnel order, the visibility structure is really no different than the visibility structure of a simple polygon *without* holes. To make this intuition more formal, we begin with a definition. Consider a pair of apexes $q < r$ that are consecutive in the funnel order of the edge (x, y). Define the *hourglass* of q and r to consist of the edge (x, y), the upper chain from r to y, the line segment (r, q), and the lower chain from q to x (see Fig. 9).

LEMMA 5.1. (i) *The four parts of an hourglass do not intersect each other except at their endpoints, and thus they define a closed simple polygon.*

(ii) *The interior of the region bounded by the hourglass is empty, that is, it contains no vertices or edges of P.*

Proof. To prove (i) consider the upper tree for edge (x, y). Since q immediately precedes r in funnel order, by Lemma 3.2, r is the clockwise postorder successor of q in the upper tree. Thus either r is the parent of q in the upper tree or else r is the furthest counterclockwise leaf in the subtree rooted at the clockwise sibling of q. If r is the parent of q in the upper tree, then the hourglass degenerates into the funnel for q, and both parts of the lemma follow immediately. Thus assume that r is not the parent of q, and let s denote the parent of q in the upper tree (see Fig. 9). The upper chain from r to y passes through s. By the clockwise and counterclockwise turning natures of the lower and upper trees, respectively, the line passing through q and s separates the upper chain from r to y from the lower chain from q to x; thus these portions of the hourglass's boundary do not intersect (except at their endpoints). This line also separates the segment (q, r) from the lower chain passing from q to x, implying that these parts of the hourglass boundary do not intersect. A symmetric argument (applied to the lower tree) shows that segment (q, r) does not intersect the upper chain from r to y. Finally, since all these structures lie within P, none of them intersects the edge (x, y).

To show (ii), consider the region R bounded by the portion of the upper chain from r to s, the segment (q, s), and the segment (q, r). Clearly, the region R, together with the interior of the funnel for q, subdivide the interior of the hourglass into disjoint regions. Because q and r are consecutive in the funnel order, there can be no vertices in the interior of R. Furthermore, there can be no edges of P in the interior of R since, in the absence of vertices in the region, such an edge would have to cross either the segment (q, s) or else the upper chain from r to s, but these are formed entirely from

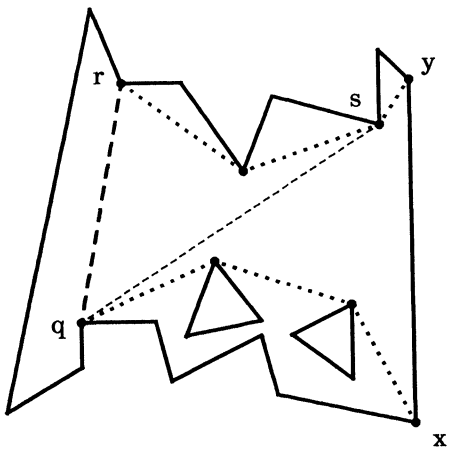

FIG. 9. *The hourglass defined by two consecutive apexes.*

visible segments. This implies that the interior of R is empty. Fact (ii) follows immediately by the emptiness of the funnel for q. □

We will also need the observation that there is consistency between the funnel ordering for edge (x, y) and the new edges.

LEMMA 5.2. *Let (x, y) be an edge of a polygonal domain P, and let v be a point external to P forming an empty triangle with (x, y). Let P' be the polygonal domain which results by replacing the edge (x, y) with the two edges (x, v) and (v, y). Let u and w be two apexes of* FNL(x, y) *such that u precedes w in funnel order.*

(i) *If u and w are both visible from v in P', then u precedes w in clockwise order about v from x to y.*

(ii) *If u and w are not visible from v in P', but both are visible from the lower edge (x, v), then u will precede w in the funnel order of (x, v) (as apexes in* FNL(x, v)*).*

(iii) *If u and w are not visible from v in P', but both are visible from the upper edge (v, y), then u will precede w in the funnel order of (v, y) (as apexes in* FNL(v, y)*).*

Proof. Assertion (ii) holds because in this case the lower chains for all such funnels are unaffected, and thus the tree relationships are preserved. Assertion (iii) holds because in this case, the upper chains for such funnels are unaffected, and thus the tree relationships are preserved (using Lemma 3.2).

To prove (i), we consider the two ways in which u can precede w in the lower tree. If u is an ancestor of w in the lower tree, then the lemma follows immediately from the convexity of the lower chains and the fact that w is visible from v. Otherwise, since u precedes w in funnel order, they share a common ancestor u'' in the lower tree, and the lower chain passing from x to w diverges clockwise from the lower chain from x to u at u''. This implies that the first segment on the path from u'' to w passes into the interior of the funnel for u. Since the funnel for u is empty, this segment must intersect the upper chain for u at some point z (not necessarily a vertex). Since u is visible from v, all of its upper chain is visible from v, and all the points on this upper chain lie clockwise from u about v. Thus z lies clockwise from u. By the convexity of the lower chains, and the fact that w is visible from v, w lies clockwise from z and hence clockwise from u about v. □

We now return to the description of the SPLIT procedure. SPLIT is a recursive procedure that is called under the following conditions. We are given the enhanced visibility graph for a polygonal domain P, an edge (x, y), and a point v external to P forming a triangle with (x, y). Throughout the description, P, x, y, and v will remain constant. We are also given a funnel apex u that is visible from v. Let w be the parent of u in the upper tree. By Lemma 3.2, w follows u in funnel order. By the convexity of the upper chain, it follows that w is also visible from v. We assume that all of the visible segments from (v, x) to (v, u) in clockwise order about v have been added to the visibility graph but that none of the visible segments after this have been added. Let FNL$[u, w]$ denote the subsequence of FNL(x, y) that contains all the funnels (in funnel order) between u and its parent w, noninclusive. (Note that the elements of FNL$[u, w]$ have edge (x, y) as their base, not the segment (u, w).) It is easy to see that, since w is the parent of u in the upper tree, the upper chain of every funnel in FNL$[u, w]$ passes through w (although the lower chain of every funnel in FNL$[u, w]$ need not pass through u).

Recall that funnels are not stored explicitly as upper and lower chains, but rather we only store the first segment of the lower chain and extract all other segments from traversals of the enhanced visibility graph. Thus as segments are added to the enhanced visibility graph, the structure of the funnels changes. With this in mind, on return from the call SPLIT(u, w), the following tasks will be completed.

(1) All the visible segments between v and visible apexes of FNL$[u, w]$ are added (each addition will, in effect, split some funnel into two funnels),

(2) FNL$[u, w]$ is split into two funnel sequences, those with apexes visible only to the lower edge (x, v) and those with apexes visible only to the upper edge (v, y). The first set of funnels are concatenated onto the end of FNL(x, v) and the second set is concatenated onto FNL(v, y). (Note that the addition of the visible segments in (1) implies that all funnels in FNL$[u, w]$ will be in either one class or the other.)

FNL(x, v) and FNL(v, y) are initialized to empty. The algorithm proceeds by first creating the visible segment (v, x), then calling SPLIT(x, y), which does the bulk of the work, and finally adding the visible segment (v, y). The fact that SPLIT will encounter visible pairs in clockwise order about v together with Lemma 5.2 implies that the final order of these newly formed sequences will be correct. We now give an annotated description of the procedure. Throughout the description, unless otherwise noted, all funnels and the lower and upper trees belong to FNL(x, y). Although we will often ignore the distinction between apexes and vertices in the description below, recall that every apex is represented by the first edge of its lower chain.

PROCEDURE SPLIT(u, w):

(1) We begin by searching for the last funnel apex q after u in funnel order whose visibility of v is blocked by u. The path to q in the lower tree may visit many vertices that are invisible from v, so we seek a more efficient route. Our method instead locates the successor r of q in funnel order (see Fig. 10). Since we have added the visible segment (u, v), this segment is the first segment of an apex in the lower tree of FNL(v, y). Let u' be the extreme clockwise child of this apex in the lower tree for (v, y). This is the most clockwise child of u whose visibility of v is blocked by u.

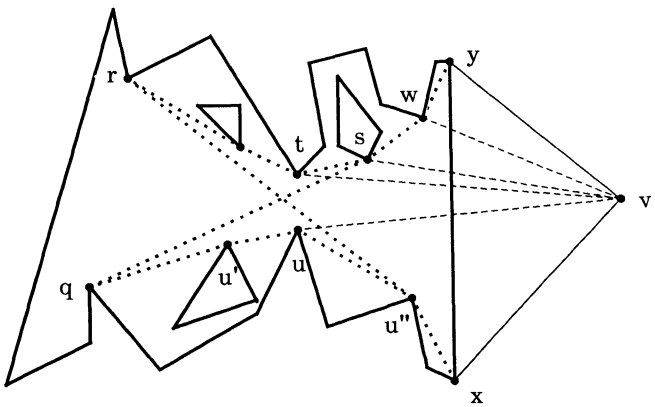

FIG. 10. *Searching for a new visible apex.*

(a) If u' is undefined, because the apex associated with (u, v) is a leaf in the lower tree of FNL(v, y), then it follows that there is no apex of FNL(x, y) whose visibility of v is blocked by u, and so we can take q to be u and r to be the successor of u in funnel order, and continue with step (2).

(b) Otherwise let us think of u' (represented by the segment from u' to u) as an apex in FNL (x, y). Let S denote the set of apexes who are descendents of either u' or of its siblings in the lower tree lying counterclockwise of u'. These apexes will be consecutive in FNL (x, y), starting from just after u to q. These are exactly the apexes whose visibility of v is blocked by u. The first apex r immediately following S in the funnel order of FNL (x, y) will be the postorder successor of u' (ignoring the ancestors of u') in a clockwise traversal of the lower tree of (x, y).

The apex r can be found by the following loop which walks along the lower tree of (x, y) towards the root. Let $u'' \leftarrow u$. Throughout the loop we will maintain the invariant that u'' is the parent of u'. While u' is the extreme clockwise child of u'', let $u' \leftarrow u''$ and let u'' be assigned its parent in the lower tree of (x, y). On exit of this loop, let r be the next clockwise sibling of u' in the lower tree. (Since $u \neq y$, such a successor will eventually be found.) Take q to be the predecessor of r in funnel order.

(This traversal toward the root of the lower tree of (x, y) is complicated technically by the fact that the tree has been altered during previous calls to SPLIT by the addition of visible segments from these vertices to v. However, the data structure described in § 7 has no difficulty ignoring these added segments and taking their clockwise neighbors instead.)

(2) If $q \neq u$, all the funnels following u up to q are known to be hidden from v, but can see the edge (v, y). Split the list FNL $[u, w]$ just after u and up to and including q, yielding the sublist of apexes a such that $u < a \leqq q$ (in funnel order). Concatenate this sublist to the end of FNL (v, y).

(3) Let s be the parent of q in the upper tree. The following properties relating r, s, and w are now relevant. These are proven later in Lemma 5.3.
   (a) Both w and s lie on the upper chain from r to y so that s lies between r and w (inclusive) on this chain.
   (b) The set of apexes on the upper chain from r to y that are visible from v form a contiguous subchain whose last element is either r or an apex t such that the line passing through v and t is tangent to the upper chain.
   (c) The segment (v, t) is the next visible segment after (v, u) in clockwise order about v.
   (d) The apex s is visible from v; that is, s lies between t and w on this upper chain.

Property (d) is key to the procedure since it implies that we have "jumped" from one visible vertex u to another visible vertex s in essentially constant time. The other properties are used to help locate the intermediate visible vertices.

The vertex that we are really interested in finding is the vertex t, which closes off the pocket started by u. Unfortunately, our search procedure only gives us s, a visible ancestor of t, in the upper tree. It would be tempting to simply search for t at this point, but in order to maintain our complexity bounds, we must make each piece of work pay off with the discovery of a new visible

segment. The remainder of the procedure "mops up" the pockets of visibility between t and w.

(4) Traverse the upper parent chain from s to w, and then backtrack along this chain from w back through s and towards r. (Backtracking is done by stacking the vertices visited from s to w and then popping the stack, while the traversal from s towards r is done by selecting the next clockwise segment in the upper tree following the segment (s, q) and then continuing along the extreme counterclockwise child of each succeeding apex. Since r is the next vertex in the upper tree following q in postorder, this process will eventually terminate at r if allowed to.) This traversal continues until reaching r or the last apex t that is visible from v. (The apex t is a point of tangency on the upper chain with respect to v (see Fig. 10).) From properties (3)(b) and (3)(d) above it follows that all of the apexes visited by these traversals are visible from v. Let $t_0, t_1, \cdots, t_k$ denote the apexes visited by this traversal in reverse order so that $t = t_0$ and $w = t_k$.

(5) The counterclockwise turning of the upper chains implies that every apex a, $r \leqq a < t$ in the funnel order, will have its visibility from v blocked by t, but each apex will be visible from the lower edge (x, v). If $r = t$, then this sublist is empty, otherwise split FNL$[u, w]$ just before r and just before t and concatenate this sublist to the end of FNL(x, v).

(6) By Lemma 5.2 the apexes $t = t_0, t_1, \cdots, t_k = w$ are given in clockwise order about v, are all visible from v, and in each case, t_i is the parent of t_{i-1} in the upper tree. It is easy to see that FNL$[u, w]$ consists of the funnels between u and t, which have already been processed, and the concatenation of FNL$[t_{i-1}, t_i]$ for $i = 1, 2, \cdots, k$ (including also the visible segments (v, t_i)). Thus the preconditions of the SPLIT procedure apply. For i running from 1 to k do the following.

   (a) Add the visible segment (v, t_{i-1}), thus effectively splitting the funnel with apex t_{i-1} into two funnels, a lower funnel whose base is edge (x, v) and an upper funnel whose base is edge (v, y).

   (b) Concatenate the lower funnel to the end of FNL(x, v) and concatenate the upper funnel to the end of FNL(v, y).

   (c) Call SPLIT(t_{i-1}, t_i). This will find all the visible apexes between t_{i-1} and t_i and will append all funnels to either FNL(x, v) or FNL(v, y) as appropriate.

The only nontrivial observations needed to establish the correctness of SPLIT are the properties mentioned in step (3).

LEMMA 5.3. *Properties* (a), (b), (c), *and* (d) *listed in step* (3) *of the above algorithm are all true.*

Proof. Since q and r are consecutive in funnel order, where q precedes r, we can apply Lemma 5.1 to the hourglass of q and r. As in that lemma, if r is the parent of q in the upper tree, then the hourglass degenerates into a funnel, and $s = r$ and is visible from v. The lemma follows immediately from basic funnel properties. It was shown in the proof of Lemma 5.1 that s lies between r and y on the upper chain. We show that s is between w and r. By Lemma 5.1 and the convexity of the upper and lower chains, since u is an ancestor of q on the lower chain, the parent of u, namely w, is an ancestor of the parent of q, namely s, on the upper chain. This establishes (3)(a). Property (3)(b) is a simple consequence of Lemma 5.1 and the convexity of the chains.

We argued earlier that all apexes strictly after u and up to q are hidden from v. To prove (3)(c) we first argue that all apexes starting with r, and up to but not including t, are also hidden from v. Property (3)(c) will then follow from Lemma 5.2, because t will be the next visible vertex in funnel order. If $r = t$, then this sequence of apexes is empty and the claim is trivially true. Otherwise the line passing through v and u is tangent to the lower chain from q to x (or possibly $q = u$). The apex r lies on the opposite side of this line because r's visibility of v is not blocked by u. Since $r \neq t$, t is a point of tangency with respect to v along the upper chain. By extending the segments (v, u) and (v, t) through u and t, respectively, to the boundary of P, we have a wedge that separates q from r. This wedge contains no apexes in its interior, for otherwise q and r would not be adjacent in funnel order. Since t is an ancestor of r in the upper tree, all the successors of r up to, but not including, t are descendents of t in the upper tree (because funnel order corresponds to a postorder traversal of the upper tree), and it is easy to see that t is a point of tangency with respect to v for the upper chains of all of these successors. Thus all these apexes are hidden from v, implying that t is the next visible apex.

To prove property (3)(d) we claim that s lies on the portion of the upper chain from r to y which is visible from v. Since this chain is convex and t is a point of tangency, this means that we must show that s lies on the portion of this upper chain from t to y. Suppose that s were to lie in the invisible portion of the upper chain from r to t. Consider the upper tree edge from q to s. Because this segment is tangent to the upper chain from r to y (and is directed so that its extension through s would stab the segment (x, y)) it would follow that q lies clockwise from t with respect to v. However, by our construction, q's visibility of v is blocked by u (or q equals u), so q lies counterclockwise of u with respect to v. This leads to a contradiction because we have just shown that t is clockwise of u with respect to v. □

Ignoring the time needed to manipulate the underlying data structure (which we will show to be $O(E)$ in § 7), the algorithm's running time is proportional to the number of visible segments added.

LEMMA 5.4. *Assuming that the graph is represented as an enhanced visibility graph, the running time of* SPLIT (x, y) *is proportional to the number of visible segments added to* v.

Proof. The procedure performs essentially only local traversals of the enhanced visibility graph, by walking around either the lower or upper trees for the edge (x, y). As mentioned in Lemma 4.1, these traversals can be performed in constant time assuming that the graph is represented as an enhanced visibility graph.

Let E_v denote the number of visible segments added to v during the call SPLIT (x, y). To show that SPLIT runs in $O(E_v)$ time note that the first argument to SPLIT is always an apex visible from v, and since successive calls are to apexes in further funnel order, SPLIT is never called with this same first argument twice. Thus, the number of recursive calls is at most E_v. The procedure contains only two loops. The first loop appears in step (1)(b) when the lower chain is searched starting from u for the vertex u'' that is the parent of r in the lower tree. Each apex visited in this loop is visible, and we claim that, with the exception of the apex u'', none of these apexes will be visited twice by this loop. The reason is that all subsequent executions of this loop will begin searching starting from some apex that comes after (or is equal to) the apex t in funnel order. Since t is an ancestor of r on the upper chain, t's ancestors on the lower chain will be ancestors of the parent of r, namely u''.

The second loop appears in step (4) where the upper chain from s back to w is traversed and then retraversed to t. All of the apexes visited in this process are visible

by Lemma 5.3, and a recursive call is made for each such apex (except w), and so the cost of this step cannot exceed $O(E_v)$ altogether. □

6. The overall algorithm. Finally we describe how to use the SPLIT procedure to compute the enhanced visibility graph for a polygonal domain P. The problem reduces to that of incorporating a new vertex v into a triangulated region P resulting in an enlarged triangulated region P'. The difference between this process and the problem that SPLIT solves is that SPLIT incorporates exactly one new triangle into P, and in the plane-sweep triangulation we incorporate one or two sequences of triangles whose bases form an inward-convex chain with respect to v by Lemma 2.1.

For each of these sequences of triangles, let $u_0, u_1, \cdots, u_m$, be vertices on the inward-convex chain that are visible from v. We consider three cases.

(1) If the sequence is empty, v cannot see any vertex on P, implying that there is no change in the visibility graph except the inclusion of the isolated vertex v. This occurs in the plane sweep whenever a vertex is inserted whose local neighborhood with respect to P lies to the right of the vertical line passing through v.

(2) If the sequence contains one vertex u_0, then v can see only u_0 on P (locally), implying that v cannot see any other vertices within P. Thus the only change in the visibility graph is the inclusion of the edge (v, u_0). This will be the case, for example, for a vertex following case (1). We will think of this single edge as consisting of two oppositely directed edges that bound a polygon with zero area. The only funnels are degenerate funnels.

(3) Otherwise, the sequence contains at least two vertices forming an inward-convex chain with respect to v. In the rest of the discussion, we consider this case.

Each triple (u_{j-1}, u_j, v) forms a triangle (see Fig. 11). The edges (u_0, v) and (v, u_m) are on the boundary of P'. Our objective is to compute FNL (u_0, v) and FNL (v, u_m) (for P'). Assume inductively that we have already computed FNL (u_{j-1}, u_j) for P for each of the edges (u_{j-1}, u_j) on the chain (since this will be a part of the representation of the enhanced visibility graph for P). For each such edge and vertex v, call the SPLIT procedure. This splits FNL (u_{j-1}, u_j) into two funnel sequences, one for the lower edge (u_{j-1}, v), which we call L_j, and the other for the upper edge (v, u_j), which we call U_j. In the process, SPLIT also adds all the visible segments from v passing through the edge (u_{j-1}, u_j). Although L_j consists of funnels for the polygon P whose common base is the edge (u_{j-1}, v), we can think of them as funnels for P' whose

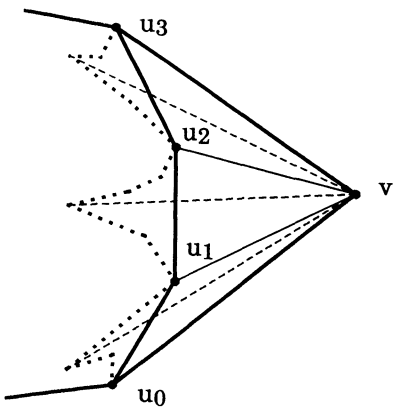

FIG. 11. *Joining a vertex to an inward-convex chain.*

common base is the edge (u_0, v). Similarly, U_j can be thought of as a sequence of funnels for P'. In order to form the desired funnel sequences for P' we appeal to the following lemma, which establishes that we can obtain this funnel sequence by concatenating the intermediate sequences.

LEMMA 6.1. *Consider a point v external to a bounded polygonal domain P and an inward-convex chain of vertices $u_0, u_1, \cdots, u_m$ $(m \geqq 1)$ on the boundary of P visible from v. Let P' be the polygon obtained by replacing this chain with the edges (u_0, v) and (v, u_m). Then*

(i) *FNL (u_0, v) in P' is equal to the concatenation of L_j for $1 \leqq j \leqq m$, followed by the trivial funnel whose apex is v.*

(ii) *FNL (v, u_m) of P' is equal to the concatenation of the trivial funnel whose apex is v followed by U_j for $1 \leqq j \leqq m$.*

(iii) *The computation of L_j and U_j does not affect the computation of L_i and U_i for any $i \neq j$.*

Proof. We first prove (i), and (ii) follows by a symmetric argument (together with Lemma 3.2). Consider the lower tree for FNL (u_0, v). Each vertex u_j is visible from v and hence is the apex of a funnel for edge (u_0, v) whose lower chain consists of $u_0, u_1, \cdots, u_j$ and whose upper chain consists of the single segment (u_j, v). To see that this forms a funnel, observe that the chain $u_0, \cdots, u_j$ is inward-convex with respect to v, and (u_0, v) is an edge of P'. In general, if C is a path in the lower tree for edge (u_{j-1}, u_j) in P, then the concatenation of $u_0, u_1, \cdots, u_{j-1}$ with C forms a chain in the lower tree for edge (u_0, v) in P'. Conversely, other than the segment (u_0, v) (corresponding to a trivial funnel), every path in the lower tree for edge (u_0, v) is of the form $u_0, u_1, \cdots, u_{j-1}$ followed by some chain C in the lower tree for edge (u_{j-1}, u_j). Hence, every funnel in L_j is extendible to a funnel of (u_0, v) and the funnel order within L_j is preserved in the extension. Since u_{j-1} is the child of u_j in the lower tree for (u_0, v), L_{j-1} precedes L_j in the funnel order for (u_0, v). This implies that FNL (u_0, v) is the concatenation of FNL (u_{j-1}, u_j) (which is L_j) for $1 \leqq j \leqq m$, followed by v.

To prove (iii) we first note that there are two things that might go wrong. First, when calling SPLIT, the list FNL (u_{j-1}, u_j) is destroyed to form L_j and R_j. However, SPLIT does not access any funnel sequences other than the one that it is decomposing, and since funnel sequences are disjoint, one decomposition does not affect another one. The second thing that may occur is that SPLIT adds visible segments from v to some of the vertices in P in order to form the visibility graph for P'. It might be that by adding these segments, we would alter the structure of a funnel in some other funnel sequence (since we use the visibility structure to traverse the funnel trees). Observe that it may be the case that this visible segment intersects the interior of a funnel for some other base edge, or even for this base. The key is that this visible segment does not alter the set of funnels or the funnel structure for any other base edge. To see this, consider the apex for a funnel located at some vertex v belonging to some other base edge e. As shown earlier, the first visible segments of the upper chain and lower chain for this apex define a wedge whose apex is v such that all rays emanating from v intersect the boundary of P first at the base e. The only way that the newly added segment could affect the structure of this funnel would be if the newly added visible segment is incident upon v and lies within this wedge. However, the fact that the new visible segment first intersects the base edge (u_{j-1}, u_j) implies that it cannot lie within this wedge. By applying this argument to every funnel of the lower tree for base edge e, we see that the newly added visible segment cannot alter the lower tree for e, and hence it cannot alter the funnel structure for e. □

Thus, the overall algorithm for building the enhanced visibility graph of P follows.

(1) Compute the plane-sweep triangulation of P by forming the triangulated polygons with holes $P_1, P_2, \cdots, P_n$. The enhanced visibility graph of P_0 is empty. For k running from 1 to n, repeat steps (2) through (4).
(2) When v_k is added to the triangulation it is connected to either one or two inward-convex chains of vertices on the boundary of P_{k-1}. For each such chain $u_0, u_1, \cdots, u_m$, perform steps (3) through (4).
(3) If the chain has length zero, then simply add the isolated vertex v_k to P_{k-1} forming P_k. If the chain has length one, then add the vertex v_k and the visible segment (v, u_0) to P_{k-1}, forming P_k.
(4) If the chain has length two or greater, call SPLIT on the polygon P_{k-1} with each edge (u_{j-1}, u_j) and vertex v_k (for $1 \leqq j \leqq m$) forming L_j and U_j. Concatenate the L_j's together with the trivial funnel whose apex is v_k and whose base is (u_0, v_k) to form FNL (u_0, v_k). Concatenate the trivial funnel whose apex is v_k and whose base is (v_k, u_m) together with the U_j's to form FNL (v_k, u_m). From these we have the enhanced visibility graph for P_k.

The running time of the complete visibility graph algorithm is proportional to the sum of the times to:

- compute the plane-sweep triangulation, which we showed to be $O(n \log n)$, plus the number of edges in the triangulation, which is $O(n)$; and
- the time needed to call the procedure SPLIT for each triangle of the triangulation, which we will show to be $O(E)$ in the next section (where E is the number of visible segments in the visibility graph).

7. Data structure. The only detail omitted in the previous sections is how the operations REV, CW, CCW, CX, and CCX are implemented. An earlier version of this paper used finger trees to implement these operations [4]. In this version we use a simpler data structure based on a data structure for the set Split-Find problem [3].

To implement the operations of CW and CCW, all that is needed is a doubly linked adjacency list for each vertex such that the entries are sorted in angular order about each vertex. To implement REV, we cross index entries for oppositely directed edges. Note that when inserting new visible segments in the SPLIT procedure, we always have access to the clockwise neighbors of the new segment (because the segments are always inserted into the middle of a funnel apex, and each apex is represented by the first segment of the lower chain). Thus updates can be performed in $O(1)$ time.

To compute the boundary extensions CX and CCX we will need to make use of the following observations about the way in which visible segments are added to this structure. Every vertex v has two phases during which segments are added to it. Phase A occurs when we are incorporating the new vertex v into the visibility graph in the SPLIT procedure. All the visible neighbors of v discovered during this phase have already been visited (have lower x-coordinates) and have already been incorporated into the visibility graph. As observed in the earlier sections, the visible neighbors added during this phase are added in clockwise order about v. Phase B neighbors arise when a vertex u whose x-coordinate is higher than v's is being incorporated into the visibility graph, and the SPLIT procedure applied to u discovered that some funnel whose apex is at v can see u. We have no control over the order (about v) in which these neighbors appear. All phase A neighbors have been added before any phase B neighbors are added.

A vertical line passing through v divides the plane into two halfplanes; the left halfplane contains the phase A visible neighbors and the right halfplane contains the phase B visible neighbors. We will maintain the segments of each phase in clockwise angular order about v, and we make the convention that there are imaginary vertical

segments, so that each segment has a predecessor in this order. Extend each phase B visible segment to a line passing through v. These linear extensions subdivide the left halfplane into a set of wedges about v. These wedges divide the phase A segments into disjoint intervals of segments. Each interval is associated with the phase B segment whose extension is the nearest counterclockwise extension to (immediately preceding) the interval, and each phase B segment is associated with the first nonempty interval that lies clockwise from (immediately after) its linear extension (see Fig. 12).

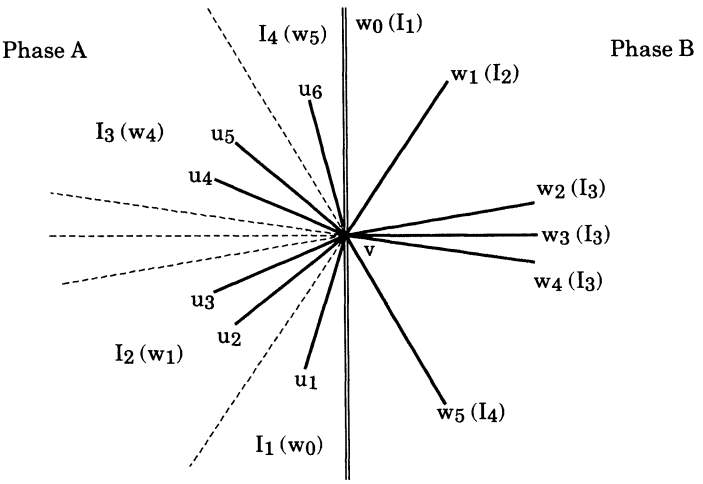

FIG. 12. *Visible neighborhood of a vertex.*

By maintaining this interval partition of the phase A neighbors, we claim that we can compute the extensions CX and CCX. Define the clockwise extension *candidate* of segment (u, v) to be the visible segment with the smallest clockwise angle greater than 180 degrees with respect to (u, v). The candidate differs from the true clockwise extension in that the clockwise extension may not be defined if one of the two boundary edges of P intersects v locally through this angular sweep. Clearly, it can be tested in constant time whether the clockwise extension candidate is the true clockwise extension. A similar definition applies to the counterclockwise extension candidate. If (u, v) is a phase A segment, then its counterclockwise extension candidate is the extension edge associated with the interval containing this segment, and the clockwise extension candidate is the clockwise neighbor of this segment. If (w, v) is a phase B segment, then the clockwise extension candidate for this edge is the first segment in the interval associated with this segment and the counterclockwise extension candidate is the last segment in the previous interval.

From this, it is clear that maintaining extensions can be reduced essentially to the problem of maintaining a partition of the phase A visible segments of v into a set of intervals, where the intervals are defined by the linear extensions of phase B visible segments of v. Let m_A denote the number of v's visible neighbors in A. Since the phase A neighbors are added in clockwise order, we can easily associate them with the set of integers $S = \{1, 2, \cdots, m_A\}$. Observe that this is just a ranking of the visible segments in order of decreasing slope. This is done on completion of the SPLIT procedure when v is incorporated into the visibility graph.

These integers are stored in a data structure developed by Gabow and Tarjan for processing Split-Find operations [3]. The Split-Find data structure (not to be confused

with the procedure SPLIT) is designed to process an intermixed sequence of the following two operations, which are seen to be a reversal of the familiar Union-Find operations:

Find(i): Return the name of the set containing i.

Split(i): Split the set containing i into two sets, one containing all integers less than or equal to i, and the other containing all integers greater than i.

Given an initial set of size a, a sequence of b Splits and Finds can be processed in total time $O(a+b)$. In addition, each Find runs in constant time.

As mentioned earlier, the Split-Find data structure is initialized to contain the integers S associated with the phase A visible segments as soon as the SPLIT procedure has completed. Before describing the processing of the phase B visible segments, there is one operation which we will need to discuss which is not supported directly by the Gabow and Tarjan data structure. When a new phase B segment is discovered, we need to find the counterclockwise extension candidate, that is, the next larger phase A segment in slope order. Note that this is not the same as a Find operation because Find assumes that the exact index of the split point is known. We assume that the slopes of the phase A segments are stored in an array sorted by decreasing slope. This order is available to us without sorting because the visible segments are added in slope order.

To update the structure when a new phase B visible segment (v, w) is added, recall that we know the existing phase B segment, say (v, w'), immediately preceding this segment in clockwise order about v. The phase A interval I associated with w' will be the interval split by the extension of the new segment. To locate the counterclockwise extension of (v, w), we perform a *dovetailed doubling search* starting at each end of the interval I. This is done by locating the endpoints of the interval I in the slope array, and performing two one-sided doubling searches starting in from opposite ends of the interval, dovetailing the operations of the two searches into an interleaved sequence. (Observe that this is essentially a simple implementation of the search performed by finger search trees.) It follows that the time required to locate the clockwise extension is proportional to the logarithm of the distance to the nearer of the ends of the interval.

If the counterclockwise extension of (v, w) is before the first segment of I, then we associate (v, w) with I and do not split the interval. If (v, w) is the extension immediately preceding I, then we update I's associated phase B segment. If the counterclockwise extension of (v, w) is the last segment in I, we associate (v, w) with the successor interval of I. Each such trivial search requires constant time, so overall their running times are bounded by $O(m_B)$, where m_B is the number of phase B visible segments incident upon v. Otherwise, we apply the dovetailed search procedure described above to locate the counterclockwise extension of (v, w). Let us say that the index of this extension is i. We call Split(i), associating the new interval of elements that are greater than i (clockwise from u_i) with (v, w). It follows that when applied to an interval of size m_A, the asymptotic running time of this algorithm satisfies the recurrence

$$T(m_A) = \max_{1<k<m-1} T(k) + T(m_A - k) + \min(\log k, \log(m_A - k)),$$

whose solution is $O(m_A)$ (see [12, p. 185]). Combining this with the $O(m_B)$ cost for the trivial finds implies that the total time spent searching for extensions about the vertex v is $O(m_A + m_B)$. Summed over all vertices, the total running time of the searches is $O(E)$.

Each CX and CCX operation performs one Find operation (observe that no Find is needed on a phase B segment, since we simply access the first or last segment in the appropriate interval). Thus each CX or CCX operation requires $O(1)$ time in our data structure, and hence $O(E)$ time overall, since our algorithm performs this many primitive operations. Each Split arises when a visible segment is added in phase B, of which there are at most m_B. Thus the total amount of time spent in the Gabow and Tarjan data structure processing the Splits is $O(m_A + m_B)$, which again is $O(E)$ when summed over all vertices.

8. Concluding remarks. We have given an $O(n \log n + E)$ algorithm for constructing the visibility graph of a set of polygonal obstacles in the plane. The construction is based on the notion of funnels, funnel sequences, and upper and lower trees, which have arisen in various forms in the study of visibility and shortest paths in polygons. These notions are combined with a novel method of traversing the visibility graph utilized in the procedure SPLIT. Together with a variation of Dijkstra's algorithms that runs in $O(n \log n + E)$ time, this provides a shortest-path algorithm in the midst of polygonal obstacles whose running time is dependent on the size of the visibility graph.

The principal drawback of our algorithm is the complexity of its implementation, particularly due to the extraction of the tree traversal primitives from the enhanced visibility graph. As an implementation note, there is a simpler data structure for the Split-Find problem that runs in $O(m \log^* n)$ time [7]. Although this leads to a theoretically slower algorithm, $O(n \log n + E \log^* n)$, it is likely that the simpler version will run faster for all reasonable input sizes. Another interesting issue is that the algorithm may need to store $O(E)$ segments at every intermediate stage. Overmars and Welzl's $O(E \log n)$ visibility graph algorithm, although inferior with respect to asymptotic complexity, requires only $O(n)$ working storage [14]. The need to store the complete visibility graph at every stage of the algorithm seems inherent in our approach.

Other sorts of visibility graphs are easily derivable from this algorithm. It is a fairly simple enhancement to the algorithm to label each funnel apex with the unique edge that can be seen by looking out from the apex through the funnel. From this the vertex-edge weak visibility graph can be derived (where a vertex and edge are adjacent if the vertex can see at least one point of the edge). The visibility polygon of a vertex can be constructed in $O(n)$ time. The edge-edge weak visibility graph can also be derived (where two edges are adjacent if they contain points which are mutually visible) since two edges e_1 and e_2 are weakly visible if and only if there exist vertices u and v such that the funnel apex whose lower chain begins with the segment (u, v) sees edge e_1, and the funnel apex whose lower chain begins with edge (v, u) sees edge e_2. Although the running time of the algorithm is dependent on the size of the standard visibility graph E, and not on the size of the edge-edge weak visibility graph E_W, it can be shown that these two quantities are asymptotically equal. To see this, observe that the above construction implies that $E_W \in O(E)$. Any pair of visible vertices (u, v) can be associated with a weak visibility between two edges of P having u and v as endpoints. Each weakly visible pair of edges is associated with at most four such visible pairs, and so $E \in O(E_W)$ (we thank one of the anonymous referees for this observation).

In general, not all of the visibility graph is needed by the shortest-path algorithm. In general, shortest paths will travel only along the lines of tangency between the obstacles. Kapoor and Maheshwari [8] have shown that such a reduced visibility graph can be computed in $O(E_R + T)$ time, where T is the time needed to triangulate the polygonal domain, and E_R is the number of edges in the reduced visibility graph.

Acknowledgments. The authors would like to thank John Hershberger and Subhash Suri for observing that the weak visibility graphs can be derived from this algorithm, and Kurt Mehlhorn for pointing out the connection between our plane-sweep triangulation and the plane-sweep triangulation algorithm for simple polygons. We would also like to thank Joe Mitchell for his careful reading of an earlier draft of this paper. Finally, we would like to thank the anonymous referees for their careful reading and valuable suggestions, and in particular for finding a subtle error in the application of the Gabow and Tarjan data structure.

REFERENCES

[1] T. ASANO, T. ASANO, L. GUIBAS, J. HERSHBERGER, AND H. IMAI, *Visibility of disjoint polygons*, Algorithmica, 1 (1986), pp. 49–63.

[2] M. L. FREDMAN AND R. E. TARJAN, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.

[3] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, J. Comput. System Sci., 30 (1985), pp. 209–221.

[4] S. K. GHOSH AND D. M. MOUNT, *An output sensitive algorithm for computing visibility graphs*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, Los Angeles, CA, 1987, pp. 11–19.

[5] L. 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 (1987), pp. 209–233.

[6] J. HERSHBERGER, *An optimal visibility graph algorithm for triangulated simple polygons*, Algorithmica, 4 (1989), pp. 141–155.

[7] J. E. HOPCROFT AND J. D. ULLMAN, *Set merging algorithms*, SIAM J. Comput., 2 (1973), pp. 294–303.

[8] S. KAPOOR AND S. N. MAHESHWARI, *Efficient algorithms for Euclidean shortest path and visibility problems with polygonal obstacles*, in Proc. 4th ACM Symposium on Computational Geometry, Urbana, IL, 1988, pp. 172–182.

[9] D. T. LEE, *Proximity and reachability in the plane*, Ph.D. thesis and Tech. Report ACT-12, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL, 1978.

[10] D. T. LEE AND F. P. PREPARATA, *Euclidean shortest paths in the presence of rectilinear boundaries*, Networks, 14 (1984), pp. 393–410.

[11] T. LOZANO-PEREZ AND M. A. WESLEY, *An algorithm for planning collision-free paths among polyhedral obstacles*, Comm. ACM, 22 (1979), pp. 560–570.

[12] K. MEHLHORN, *Data Structures and Algorithms, Volume* 1: *Sorting and Searching*, Springer-Verlag, Berlin, 1984.

[13] K. MEHLHORN, *Data Structures and Algorithms, Volume* 3: *Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[14] M. H. OVERMARS AND E. WELZL, *New methods for constructing visibility graphs*, in Proc. 4th ACM Symposium on Computational Geometry, Urbana, IL, 1988, pp. 164–171.

[15] M. SHARIR AND A. SCHORR, *On shortest paths in polyhedral spaces*, SIAM J. Comput., 15 (1986), pp. 193–215.

[16] E. WELZL, *Constructing the visibility graph for n line segments in* $O(n^2)$ *time*, Inform. Process. Lett., 20 (1985), pp. 167–171.