Algorithmica (1989) 4: 141-155

Algorithmica
© 1989 Springer-Verlag New York Inc.

An Optimal Visibility Graph Algorithm for Triangulated Simple Polygons[1]

John Hershberger[2]

Abstract. Let P be a triangulated simple polygon with n sides. The visibility graph of P has an edge between every pair of polygon vertices that can be connected by an open segment in the interior of P. We describe an algorithm that finds the visibility graph of P in $O(m)$ time, where m is the number of edges in the visibility graph. Because m can be as small as $O(n)$, the algorithm improves on the more general visibility algorithms of Asano *et al.* [AAGHI] and Welzl [W], which take $\Theta(n^2)$ time, and on Suri's $O(m \log n)$ visibility graph algorithm for simple polygons [S].

Key Words. Simple polygon, Visibility graph, Triangulation, Shortest path, Shortest-path map.

1. Introduction. The visibility graph of a set S of n nonintersecting line segments in the plane records each pair of segment endpoints that can be connected by an open segment free of intersections with S. If we regard the segments as opaque walls, the visibility graph records, for every wall endpoint v, what other wall endpoints would be seen by an observer standing at v. As its name suggests, the visibility graph is a combinatorial graph structure; its nodes are the segment endpoints, and its edges connect mutually visible segment endpoints.

Although the best visibility graph algorithms (by Asano *et al.* [AAGHI] and Welzl [W]) find the visibility graph in $\Theta(n^2)$ time, the visibility graph itself may be as small as $O(n)$. This seems to suggest some inefficiency in the algorithms; a linear amount of work is being expended to find each visibility graph edge. It would be desirable to have an algorithm whose running time depended on the size of its output.[3]

It is possible to improve on the $\Theta(n^2)$ algorithms when S has some special property. For example, when the segments in S form a collection of k convex polygons, it is not hard to find the visibility graph in time proportional to the output size plus $O(nk \log n)$. This gives an $\Omega(n^{3/2} \log^{1/2} n)$ best-case algorithm.

This paper considers the case in which the segments form a simple polygon with no consecutive collinear edges. By convention, the visibility graph of a simple polygon P contains only the visibility edges inside P. If we need the

[1] This work was supported in part by a U.S. Army Research Office fellowship under agreement DAAG29-83-G-0020.
[2] DEC Systems Research Center, 130 Lytton, Palo Alto, CA 94301, USA.
[3] A recent algorithm of Ghosh and Mount solves this problem: it finds the visibility graph of a set of line segments in $O(m+n \log n)$ time and $O(m)$ space, where m is the size of the visibility graph [GM].

Received April 15, 1987; revised December 7, 1987. Communicated by Chee-Keng Yap.

complete visibility graph, we can construct it by computing the convex hull of P, then finding the visibility graph inside P and inside each of the bays cut off by non-polygon edges of the hull. We define m to be the number of visibility edges inside P. Because P can be triangulated, m is at least $n-3$.

A recent algorithm of Suri finds the visibility graph of a simple polygon in time $O(m \log n)$ [S]. The method uses $O(m)$ "shooting" queries: given a direction d and a point q inside P, each query asks for the first point of P intersected by the ray from q with direction d. Each query takes $O(\log n)$ time to answer [CG].

We show how to find the visibility graph of an already-triangulated simple polygon P in $O(m)$ time. The algorithm exploits the close relationship between the visibility graph and shortest paths inside P: shortest paths follow polygon edges and visibility edges; visibility edges are shortest paths between the points they connect. The algorithm computes all the shortest paths from each polygon vertex to other vertices, then picks out the visibility edges from the single-edge paths.

At first glance, this approach seems extraordinarily inefficient. Why should we compute shortest paths with many turns when only the single-edge paths are important? The answer is that the set of all shortest paths has more structure than the set of single-edge paths. Once we have found the set of all shortest paths from one vertex, we can exploit the structure to find similar sets for the other vertices efficiently.

In outline, our algorithm is the following: We begin by representing all shortest paths from some arbitrary vertex v_1 in a structure called the *shortest-path map.* From this structure we can find the vertices visible from v_1 in constant time apiece. Let v_2 be the vertex of P counterclockwise of v_1 along the boundary of P. The shortest-path map for v_2 is nearly the same as that for v_1, and we can build one from the other in time proportional to the number of differences between them. We proceed around the boundary of P, constructing the shortest-path map of each vertex from that of its clockwise neighbor. At each step we obtain all visibility edges from the current vertex.

We build the first shortest-path map using the linear algorithm of Guibas *et al.* [GHLST]. After the initial shortest-path map has been built, the rest of the algorithm runs in $O(m)$ time. Each transformation of one shortest-path map into another takes time proportional to the number of differences between them, and, as Section 4 shows, the total number of differences between adjacent shortest-path maps is $O(m)$. Since m is at least $n-3$, our algorithm takes $O(m)$ time to find the visibility graph of a triangulated simple polygon. If we are given an untriangulated polygon as input, we must first triangulate it: this takes $O(n \log\log n)$ time using the algorithm of Tarjan and Van Wyk [TV]. In this case our algorithm runs in $O(m+n \log\log n)$ time, which is slightly suboptimal when $m = o(n \log\log n)$, but is still a significant improvement over earlier methods.

2. Shortest-Path Maps. This section defines the shortest-path map, which is a representation of all shortest paths from a source vertex s to points inside P. The shortest-path map partitions the interior of P into triangular regions, each with

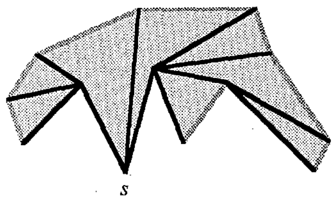

Fig. 1. The shortest-path tree is the union of the shortest paths from a source vertex s to all other vertices.

a distinguished vertex called the *apex*. The regions are chosen so that the shortest path from s to a point x passes through the apex of the region containing x.

Let us introduce some notation to help talk about shortest paths. For a point x inside P or on its boundary, we denote by $\pi(s, x)$ the unique shortest path from s to x that does not go outside P. Next we pick out a key point on the path: $lv(s, x)$ is the polygon vertex on $\pi(s, x)$ that is not equal to x but closest to it along the path (lv stands for the *last vertex* on the path). We assume that s is a polygon vertex, so $lv(s, x)$ is always well defined. For all points x inside a triangular region of the shortest-path map, $lv(s, x)$ is constant and equal to the region's apex.

Our construction of the shortest-path map begins with the ***shortest-path tree,*** which is the union, over all polygon vertices v, of the paths $\pi(s, v)$ (see Figure 1). Because there is exactly one shortest path from s to any vertex, this set of paths does indeed form a tree. The edges of the shortest-path tree partition the interior of P into regions we call *funnels.*

Each polygon edge $\overline{pq}$ has an associated funnel. The shortest path from s to p, $\pi(s, p)$, shares edges with $\pi(s, q)$ for part of its length; at some vertex a (called the *apex of the funnel*) the two paths part and proceed separately to their destinations. The two paths cannot rejoin, once separated. The funnel is bounded by the edge $\overline{pq}$ (called the *base of the funnel*) and the two *funnel paths* $\pi(a, p)$ and $\pi(a, q)$. By the definition of shortest paths, each of the funnel paths is *inward convex*: it bulges in toward the funnel region. See Figure 2 for an example of a funnel. Note that the apex of the funnel may be s, and that the funnel can degenerate into a single segment: if $a = p$, then $\pi(a, q) = \overline{pq}$.

To produce the shortest-path map from the shortest-path tree, we partition each funnel with a nonempty interior into regions where $lv(s, x)$ is constant. To do this, we extend each edge on the side of a funnel until it hits the base of the funnel (see Figure 3). Each resulting region is triangular; the *base of the region*

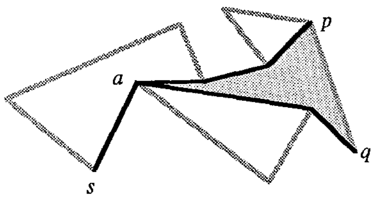

Fig. 2. The funnel belonging to edge $\overline{pq}$ is the region between the shortest paths from s to p and q. The vertex a at which the paths separate is the apex of the funnel.

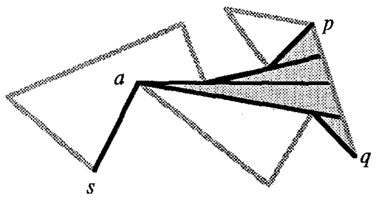

Fig. 3. Each funnel is broken into triangular regions by extending the edges on the sides of the funnel.

is a subset of the funnel base, and the *apex of the region* is the triangle vertex opposite the base. For every point x inside a region, $lv(s, x)$ is equal to the region's apex.

Although the shortest-path map partitions the interior of P into triangles, it is generally not a triangulation. Polygon vertices may lie on edges of the triangular regions. Furthermore, although the apex of each triangular region is a vertex of P, the other two vertices of the triangle need not be polygon vertices. They may instead be intersections of the funnel base with extended edges from the funnel sides.

If the shortest-path map were larger than $O(n)$, building it would be a bottleneck for our algorithm. This potential problem does not arise: both the shortest-path tree and the shortest-path map have a linear number of edges. Each node of the shortest-path tree is a vertex of P; because P has n vertices, the shortest-path tree has $n-1$ edges. Every edge of the shortest-path map in the interior of P is either an edge of the shortest-path tree (a *shortest-path edge*) or an extension of such an edge (an *extension edge*). Each edge of the shortest-path tree has at most one extension; an extension splits one base edge in two. Including the (possibly split) edges of P, a linear number of edges bound the regions of the shortest-path map.[4]

In our visibility graph algorithm, we use n different shortest-path maps, each with a different source vertex. To distinguish between different shortest-path maps, we use the notation $SPM(s)$ to refer to the shortest-path map with s as its source vertex.

In $SPM(s)$, the regions with s as their apex contain all the points inside P that are visible from s. The polygon vertices connected to s by edges of $SPM(s)$ include all the vertices visible from s. (The edges of $SPM(s)$ include the two polygon edges that connect s to its neighbors on the boundary of P, but these vertices are not visible according to our definition of visibility.)

We represent $SPM(s)$ using a standard subdivision representation, such as the winged-edge data structure of Baumgart [B] or the quad-edge data structure of Guibas and Stolfi [GS]. The data structure lets us find the edges of $SPM(s)$ with s as an endpoint in constant time apiece, which gives us the visibility edges incident to s in constant time per edge. The algorithms of Section 3 must be able to find the base of a region in constant time, so we augment the data structure

[4] By an argument that relates the number of single-edge funnels in the shortest-path tree to the number of vertices on the sides of funnels, we can show that the shortest-path map has at most $3n-6$ edges. This bound is tight.

by storing at the apex of each region a pointer to its base. When the algorithms need to find the base of a region, they already know its apex.

3. Constructing Shortest-Path Maps.

Our algorithm finds the visibility graph by constructing a shortest-path map for each polygon vertex. It first builds $SPM(v_1)$ for an arbitrary vertex v_1 using the linear algorithm of Guibas *et al.* [GHLST], then modifies the first shortest-path map to produce the others. In this section we describe the general step of the algorithm, which transforms $SPM(s)$ into $SPM(t)$ for adjacent polygon vertices s and t. We show that the construction takes time proportional to the number of differences between the two shortest-path maps. The next section shows that the number of differences, summed over all neighbor pairs, is proportional to the size of the visibility graph.

Let s and t be adjacent polygon vertices, s clockwise of t. Given $SPM(s)$, we produce $SPM(t)$ by adding and deleting edges. We change edges in the interior of P (*interior edges*) and on its boundary (*boundary edges*). The edges in the polygon interior are the important ones; when we add or delete them, we accommodate their endpoints by splitting or merging the boundary edges. The boundary edges change only when the interior edges do, and each interior edge change causes at most one split or merge of boundary edges. We therefore focus on the changes to interior edges.

Let $DIFF(s, t)$ be the set of interior edges added and deleted to transform $SPM(s)$ into $SPM(t)$. That is, $DIFF(s, t)$ is the symmetric difference between the two sets of interior edges given by $SPM(s)$ and $SPM(t)$. The following lemma characterizes this symmetric difference.

LEMMA 1. *Let s and t be adjacent vertices of P, let e be an edge of $DIFF(s, t)$, and let x be any point on e. Then any open neighborhood of x includes points that are visible from the interior of $\overline{st}$.*

PROOF. Without loss of generality, assume that e is an interior edge of $SPM(s)$ that does not belong to $SPM(t)$. The edge e separates two regions of $SPM(s)$, and hence $lv(s, a) \neq lv(s, b)$ for a and b on opposite sides of e. Because e does not belong to $SPM(t)$, any open neighborhood of x contains two points a and b on opposite sides of e but in the interior of a single region of $SPM(t)$. This implies that $lv(t, a) = lv(t, b)$. By possibly interchanging a and b, we can ensure that $lv(t, a) \neq lv(s, a)$. See Figure 4.

Now consider the shortest paths from a to s and t. Since $lv(t, a) \neq lv(s, a)$, the two paths $\pi(a, s)$ and $\pi(a, t)$ intersect only at a. If a were a polygon vertex, the region bounded by $\overline{st}$ and the two paths would be a funnel in the shortest-path tree of a; even though a is not a vertex, the inward convexity of the two paths shows that some part of the interior of $\overline{st}$ is visible from a. Any ray with source a and direction strictly in the angle formed by $lv(s, a)$, a, and $lv(t, a)$ must hit the interior of $\overline{st}$ before it hits any other part of P. □

The part of P visible from at least one point in the interior of $\overline{st}$ is a connected subpolygon, known as the *edge-visibility polygon* of segment $\overline{st}$, which we denote by $P(\overline{st})$. The preceding lemma shows that the interior edges of $SPM(s)$ and

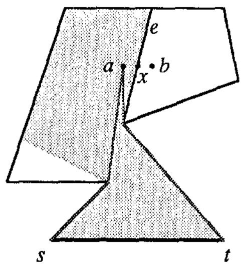

Fig. 4. Edge e belongs to $SPM(s)$ but not to $SPM(t)$, and it lies in the polygon interior. For any point x on e, there is a point a arbitrarily close to x that is visible from some point in the interior of $\overline{st}$.

$SPM(t)$ differ only inside $P(\overline{st})$ and on its boundary. The following lemma characterizes the boundary of $P(\overline{st})$.

LEMMA 2. *Let e be an edge on the boundary of $P(\overline{st})$ that is not on the boundary of P. Then e is an edge of $SPM(s)$ or $SPM(t)$. The edge has one endpoint, call it a, that lies on the paths from s and t to the other endpoint, call it b: the endpoint a is both $lv(s, b)$ and $lv(t, b)$. If a is clockwise of b on the boundary of P, then e belongs to $SPM(s)$; otherwise, it belongs to $SPM(t)$.*

PROOF. Let x be a point in the interior of e. The point x is not visible from $\overline{st}$, since it is not in the interior of $P(\overline{st})$. As noted in the proof of Lemma 1, we must have $lv(s, x) = lv(t, x)$. Let us define a to be $lv(s, x)$. We know that $lv(s, z) = lv(t, z) = a$ for any point z in the interior of $\overline{ax}$. This segment must go toward s and t from x, and hence cannot go outside $P(\overline{st})$. Since $lv(s, z) \neq lv(t, z)$ for any point z in the interior of $P(\overline{st})$, segment $\overline{ax}$ must lie on e. Thus a must be an endpoint of e. Let b be the other endpoint of e. The shortest paths from b to s and t must go through a, and so a is both $lv(s, b)$ and $lv(t, b)$.

Assume that a is clockwise of b. This means that the interior of $P(\overline{st})$ lies to the left of the directed segment $\overrightarrow{ab}$ (on the left side of $\overline{ab}$ as seen by an observer looking from a toward b). Let x' be a point just to the left of x, inside $P(\overline{st})$ (see Figure 5). Because x' is visible from $\overline{st}$, the two shortest paths $\pi(s, x')$ and $\pi(t, x')$ intersect only at x'. Vertex a is on or to the right of $\pi(t, x')$, so $lv(s, x') \neq a$. This holds for x' distinct from x but arbitrarily close to it. Since $lv(s, x) = a$, the segment $\overline{ab}$ is an edge of $SPM(s)$; it separates two regions of $SPM(s)$.

If a is counterclockwise of b, then the interior of $P(\overline{st})$ lies to the right of ab. An argument similar to the preceding one shows that $\overline{ab}$ is an edge of $SPM(t)$. □

We want to transform $SPM(s)$ into $SPM(t)$ in time proportional to the number of edges that must be changed to get from one to the other. Lemma 1 shows that no interior edges need to be changed outside $P(\overline{st})$. In the remainder of this section we discuss two dissimilar algorithms for performing the transformation. We describe the first algorithm in detail, but merely sketch the second one.

The first transformation algorithm might be called a "sweeping-path" algorithm. It moves a point p from t to s along the boundary of the subpolygon $P(\overline{st})$, the

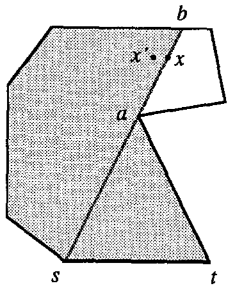

Fig. 5. Because x' is visible from $\overline{st}$ and x is not, $lv(s, x) \neq lv(s, x')$. This implies that $\overline{ab}$ is an edge of $SPM(s)$.

region in which one shortest-path map must be replaced by another. We think of the point as being connected to s and t by two rubber bands inside P, the shortest paths $\pi(s, p)$ and $\pi(t, p)$. As the point moves, it drags the two paths along with it, and the paths sweep over the subdivision inside $P(\overline{st})$. The algorithm uses the sweeping paths to change $SPM(s)$ into $SPM(t)$. The algorithm maintains the following invariant: on and to the right of $\pi(t, p)$, the subdivision represents $SPM(t)$; to the left of $\pi(s, p)$, the subdivision represents $SPM(s)$; between the two paths, there are no edges.

The algorithm updates the subdivision when p visits vertices of $SPM(s)$ or $SPM(t)$. If v is a vertex of $SPM(s)$, all the edges of $\pi(s, v)$ are edges of $SPM(s)$. When p visits v, the algorithm deletes the last edge of the path $\pi(s, v)$, unless it is also an edge of $SPM(t)$. (An edge may belong to both shortest-path maps only if it is a boundary edge of $P(\overline{st})$.) Similarly, when p visits a vertex v of $SPM(t)$, the algorithm adds the last edge of $\pi(t, v)$ to the subdivision, unless it is already an edge of $SPM(s)$. This procedure considers just once each edge that must be changed.

When the algorithm adds or deletes an edge, it may have to make other changes to the subdivision. Suppose that the algorithm deletes an edge of $SPM(s)$ when p visits v. If v is a vertex of $SPM(s)$ but not of P, then v splits a polygon edge; the algorithm must merge the two pieces back into one. Similarly, when the algorithm adds an edge of $SPM(t)$ to the subdivision, it may have to split a polygon edge: if p is not at a polygon vertex, the algorithm splits the polygon edge on which p lies. Since adding and deleting edges changes regions in the current subdivision, we must update the base pointer in each changed region. This is not hard: whenever a pointer needs to be changed, the new base is incident to p, and the region's apex is connected to p by a straight segment of $\pi(s, p)$ or $\pi(t, p)$.

How do we represent p and the two paths in a discrete algorithm? The point p does not move continuously along the boundary of $P(\overline{st})$, but in discrete jumps from one vertex of $SPM(s)$ or $SPM(t)$ to the next such vertex. Because the shortest paths to intermediate vertices on $\pi(s, p)$ and $\pi(t, p)$ are unique, the movement of p changes $\pi(s, p)$ and $\pi(t, p)$ only at the ends nearer p. Thus we can represent the two paths as stacks of polygon vertices. The stacks do not

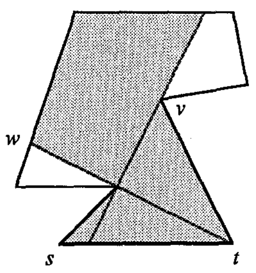

Fig. 6. $P(\overline{st})$ is bounded by polygon edges and by extension edges. The left extension edge is an edge of $SPM(t)$; the right edge belongs to $SPM(s)$.

include p: the top of the stack for $\pi(s,p)$ is $lv(s,p)$, and the top of the stack for $\pi(t,p)$ is $lv(t,p)$.

We have described the basic idea of the algorithm, as well as its data structures. We now describe the actions necessary to move p along the boundary of $P(\overline{st})$. These can be broken down into several interrelated tasks. The hardest two tasks are deciding which edge p should follow when it leaves a vertex and deciding when it should stop moving along the edge. By referring to the two shortest paths $\pi(s,p)$ and $\pi(t,p)$, the algorithm can make these decisions in constant time apiece. A simpler task is that of maintaining the stacks that represent the shortest paths. We show how to perform these tasks, taking into account the possibility of degeneracies in P. We then combine the costs of these operations to bound the algorithm's running time.

The region visible from the interior of $\overline{st}$, $P(\overline{st})$, is bounded by polygon edges and by extension edges of the types shown in Figure 6. The algorithm for following its boundary is simple: p moves counterclockwise along the polygon boundary whenever doing so leaves both shortest paths $\pi(s,p)$ and $\pi(t,p)$ inward convex. When moving along the polygon boundary would cause one of these two paths to violate inward convexity, as at vertices v and w in Figure 6, p moves along an extension edge. It moves along edges of $SPM(s)$ when moving away from $\overline{st}$ and along edges of $SPM(t)$ when moving toward $\overline{st}$, as specified by Lemma 2. In Figure 6, when p advances from v, it moves along the extension of the edge from $lv(s,v)$ through v. When p advances from w, it moves along the extension edge between w and $lv(t,w)$.

The movement of p fits in with the edge addition and deletion scheme outlined above. Whenever p moves along an extension edge of $SPM(s)$, the edge still belongs to the subdivision. When p moves along an extension edge of $SPM(t)$, the edge has just been added to the subdivision.

We can tell which edge p should follow when it leaves a vertex; we need to know when to stop its motion along the edge. There are two kinds of edges that p traverses: polygon edges visible from the interior of $\overline{st}$, and extension edges. We give separate strategies for dealing with the two types of edges.

When p moves along a polygon edge visible from $\overline{st}$, it should stop at the first vertex of $SPM(s)$ or $SPM(t)$ on the edge. It is easy to detect vertices of $SPM(s)$: they are vertices of the subdivision left of $\pi(s,p)$. Recognizing vertices of $SPM(t)$

that are not vertices of $SPM(s)$ is more difficult. We can state the problem more precisely: given a point p on the boundary of $P(\overline{st})$ such that the polygon edge counterclockwise of p is visible from $\overline{st}$, find the first vertex of $SPM(s)$ or $SPM(t)$ counterclockwise of p along the edge. Let q be the first vertex of $SPM(s)$ counterclockwise of p; it is part of the current subdivision. Points p and q belong to a single polygon edge on the boundary of $P(\overline{st})$, though neither p nor q is necessarily a vertex of $P(\overline{st})$. We need to determine whether any vertices of $SPM(t)$, excluding p, lie on $\overline{pq}$, and, if so, to find the one closest to p.

We begin by characterizing the vertices of $SPM(t)$ that lie between p and q, assuming that at least one exists. Let v be such a vertex, chosen closest to p if there are several such vertices. Vertex v is an endpoint of an extension edge e. The edge separates two regions of $SPM(t)$; v lies on the base of both. Point p is on the boundary of one of the regions, since v is the vertex of $SPM(t)$ closest to p. The apexes of the two regions lie on the line containing e. Because v is not a polygon vertex, the two apexes cannot be coincident.

Our scheme for finding vertex v needs to have $lv(t, p)$ equal to the apex of the region that has $\overline{pv}$ on its base. To guarantee this property, we move p an infinitesimal distance toward q, updating $\pi(t, p)$ and $\pi(s, p)$ accordingly. After the move, p is not collinear with the last two vertices on $\pi(t, p)$ or the last two on $\pi(s, p)$.

We split our argument into two cases, depending on the relative positions of the apexes of the regions incident to v:

In the first case, the apex to the right of e is closer to v than the apex to the left. Figure 7(a) gives an idealized view of this case, and Figure 8(a) shows a possible instance of it. (We define right and left by assuming that e is directed away from t.) Because p is an interior point of the base of the region on the right, the last two vertices of $\pi(t, p)$ lie on the segment $\overline{ab}$ of Figure 7(a). (In degenerate cases several polygon vertices may lie on $\overline{ab}$.) Hence v can be determined as the intersection of the ray through the last two vertices on $\pi(t, p)$ with the edge $\overline{pq}$.

In the second case, the apex to the left of e is closer to v than the apex to the right, as in Figures 7(b) and 8(b). Because $\pi(s, v)$ cannot lie to the left of the polygon vertex b or to the right of $\pi(t, v)$, it must go through b. Thus b is $lv(s, v)$. The region of $SPM(t)$ to the right of e has a as its apex; that is, $lv(t, p) = a$.

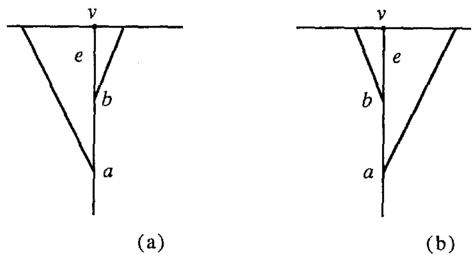

Fig. 7. The point v is a vertex of $SPM(t)$ that lies between p and q. (a) and (b) show the two configurations of $SPM(t)$ that give rise to such a vertex.

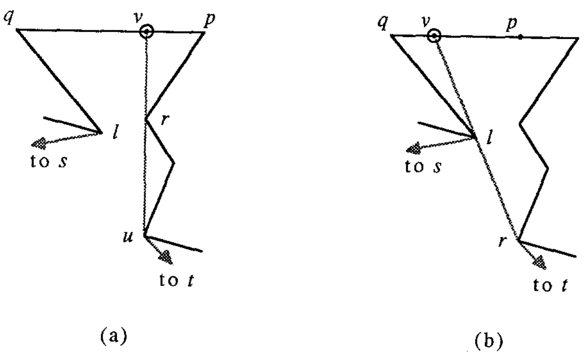

Fig. 8. The circled points are vertices of $SPM(t)$ that do not appear in the subdivision left of $\pi(s, p)$. They are detected when the algorithm tries to move p along the polygon boundary to q. (Vertices l and r are the last polygon vertices on $\pi(s, p)$ and $\pi(t, p)$, respectively. Vertex u is the predecessor of r on $\pi(t, p)$.) In (a), vertices u and r correspond to a and b of Figure 7(a). The circled vertex v is the intersection of $\overline{pq}$ with the extension of $\overrightarrow{ur}$. When p advances infinitesimally past this vertex, r is removed from $\pi(t, p)$, creating the situation shown in (b). Here r and l correspond to vertices a and b of Figure 7(b). The circled vertex v is the intersection of $\overrightarrow{rl}$ with $\overline{pq}$.

Because there are no vertices of $SPM(s)$ between p and q, we know that $lv(s, p)$ is either equal to b or to a polygon vertex in the interior of $\overline{ab}$. (Because at least part of $\overline{pq}$ is visible from $\overline{st}$, $lv(s, p)$ cannot be a. Only in the degenerate case is it not equal to b; in this case $v = q$.) Hence v can be determined as the intersection of the ray through $lv(t, p)$ and $lv(s, p)$ with the edge $\overline{pq}$.

To detect vertices of $SPM(t)$ on the edge $\overline{pq}$, we need to compute only two intersections. If the ray through the last two vertices on $\pi(t, p)$ intersects $\overline{pq}$, or the ray through $lv(t, p)$ and $lv(s, p)$ intersects $\overline{pq}$, then we move p to the intersection farthest clockwise; it is a vertex of $SPM(t)$. Otherwise we move p to q. (If $lv(t, p) = t$, the first ray is undefined and its intersection does not exist. If either ray hits $\overline{pq}$, the ray farther clockwise hits $\overline{pq}$ without hitting any other part of P first; this is because the quadrilateral determined by p, q, $lv(s, p)$, and $lv(t, p)$ is free of polygon points.) See Figure 8 for examples of these intersections.

We have described how the algorithm moves p along an edge visible from $\overline{st}$; we now describe how it moves p along an extension edge. Extension edges require care because they may have multiple polygon vertices along them, even though no edges of the shortest-path map need to be changed. This degenerate case is depicted in Figure 9. The algorithm handles extension edges specially to avoid having to visit many vertices when changing only a few edges of the shortest-path map. It uses the base pointer stored at the apex of each region. The apex is the last vertex on $\pi(s, p)$ or $\pi(t, p)$; the algorithm follows its pointer to skip over extension edges, moving directly from the first vertex to the base, or vice versa. In Figure 9, p moves directly from v to u and from w to x. If there are intermediate vertices on the extension edge (which can be checked in constant time), the edges along the extension are the same in both $SPM(s)$ and $SPM(t)$ and need not be altered. If there are no intermediate vertices, the edge can be added or deleted as necessary in constant time.

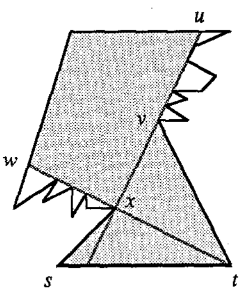

Fig. 9. In degenerate cases, multiple polygon vertices may lie on one extension edge on the boundary of $P(\overline{st})$. None of the shortest-path-map edges along the extension edge needs to be changed, so the algorithm skips over them.

The shortest paths $\pi(s, p)$ and $\pi(t, p)$ are not hard to maintain. They change only when p passes through a vertex of $SPM(s)$ or $SPM(t)$. When p passes a vertex of $SPM(t)$ of the type shown circled in Figure 8(a), then r, the last vertex of $\pi(t, p)$, is popped off the top of its stack. In degenerate cases there may be polygon vertices in the interior of $\overline{ur}$. These vertices are popped also; popping stops when p is to the right of the ray determined by the top two vertices on the stack. The stack grows when p passes a vertex like v in Figure 10. When p advances from v along edge $\overline{vq}$ and q is to the right of $\overrightarrow{rv}$, then v is pushed onto the stack for $\pi(t, p)$. In the degenerate case, $\overline{vq}$ may be an extension edge with vertices in its interior. As described above, these interior vertices are skipped over and not pushed onto the stack, since the next movement of p would pop them all back off. The stack for $\pi(t, p)$ changes only in the two cases we have described; the stack for $\pi(s, p)$ changes analogously. The two shortest paths change only in these four situations.

The algorithm runs in time proportional to the number of vertices p visits. The only vertices p visits where the algorithm does not modify the current subdivision are vertices v where $lv(s, v) = lv(t, v)$ and the edge from v to $lv(s, v)$ belongs to both $SPM(s)$ and $SPM(t)$. Such a vertex is one endpoint of an extension edge on the boundary of $P(\overline{st})$; it is invisible from $\overline{st}$, a "shadow" of a vertex that is visible from $\overline{st}$. The algorithm changes the subdivision at the visible vertex of

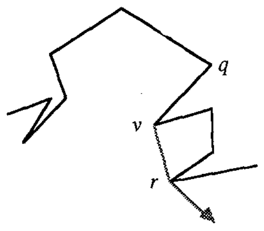

Fig. 10. When p visits v, the top of the stack for $\pi(t, p)$ is r. When p advances from v toward q, vertex v is pushed onto the stack.

which v is a shadow. Since the algorithm visits at most one shadow of each visible vertex, we have the following theorem.

THEOREM 1. *The sweeping-path algorithm computes $SPM(t)$ from $SPM(s)$ in time proportional to the number of differences between the two shortest-path maps.*

Our second transformation algorithm is based on the paradigm of depth-first search. We merely sketch this approach, which is derived from the algorithm for constructing shortest-path maps of Guibas *et al.* [GHLST]. The depth-first search algorithm uses finger search trees [GMPR], [HuM], and hence it may be harder to implement than the sweeping-path algorithm. The idea behind the depth-first search algorithm is not difficult: the algorithm of Guibas *et al.*, which runs in a triangulated simple polygon, can be modified to run using a shortest-path map in place of the triangulation. We build $SPM(t)$ by running the modified algorithm in $SPM(s)$. In essence, the algorithm is a depth-first exploration and transformation of the regions of the subdivision, starting with the region incident to edge $\overline{st}$. Because the subdivision needs to be changed only in the regions overlapping $P(\overline{st})$, the search limits itself to that subpolygon. As a consequence, the algorithm runs in time proportional to the number of differences between $SPM(s)$ and $SPM(t)$.

4. Complexity Bounds.

This section shows that each of the algorithms of the preceding section takes $O(m)$ time overall. The proof associates a constant number of edges of the shortest-path maps with each visibility edge or polygon edge; every edge that appears in a shortest-path map has an associated visibility edge or polygon edge. As the source of the shortest-path map moves around the boundary of P, edges are added to and deleted from the current shortest-path map. We show that each edge is added and deleted a constant number of times, which gives the desired bound on the running time.

It is helpful to think of shortest-path edges as being directed. The edge $\overline{pq}$ has two versions, $\overrightarrow{pq}$ and $\overrightarrow{qp}$. When $\overline{pq}$ appears in some shortest-path map $SPM(v)$, we write it as $\overrightarrow{pq}$ if p lies on the path from v to q and $\overrightarrow{qp}$ otherwise.

Edges of shortest-path maps are of three types: shortest-path edges, extension edges, and boundary edges. Some boundary edges are also shortest-path edges; those that are not are exactly the base edges of the nontrivial regions of the shortest-path map. (Recall that the funnel of edge $\overline{pq}$ may be just $\overline{pq}$, in which case the region associated with $\overline{pq}$ is trivial.) A base edge changes only when its endpoints do. Each change to a base edge is caused by a change to an extension edge, and each extension-edge change affects at most two base edges. Thus it suffices to bound the number of changes to shortest-path edges and extension edges to bound the running time of our algorithm.

We associate every shortest-path edge or extension edge with a directed polygon edge or visibility edge. Shortest paths follow visibility edges and polygon edges, so we only need to account for the extensions of funnel edges. Each funnel edge is a polygon edge or a visibility edge; we associate each extension edge with the version of its funnel edge that is directed toward it. Each directed edge has at most one extension.

The following lemma characterizes the shortest-path maps in which each directed edge appears.

LEMMA 3. *Let $\overline{pq}$ be a polygon edge or a visibility edge. The shortest-path maps in which the directed edge $\overrightarrow{pq}$ appears have sources that form a contiguous subsequence of the polygon vertices.*

PROOF. We exhibit polygon vertices l and r such that $\overrightarrow{pq}$ appears in $SPM(v)$ for all v strictly counterclockwise of l and clockwise of r in the sequence of polygon vertices, and $\overrightarrow{pq}$ appears in no other shortest-path maps. We first note that $\overrightarrow{pq}$ must appear in $SPM(p)$. Let l be the first vertex clockwise of p for which $\overrightarrow{pq}$ is not in $SPM(l)$. Because shortest paths do not cross, $\overrightarrow{pq}$ cannot appear in any shortest-path map whose source is between l and q, inclusive. A similar argument shows the existence of r and proves the lemma. □

Now consider the transformations of shortest-path maps described in the previous section. If $\overline{st}$ is an edge of P, $SPM(t)$ can be constructed from $SPM(s)$ in time proportional to the number of edges that must be changed to get from one to the other. The preceding lemma lets us bound the time needed to produce shortest-path maps for all the polygon vertices.

THEOREM 2. *Given $SPM(v_1)$ for any polygon vertex v_1, repeated application of either of the algorithms of Section 3 produces a shortest-path map for every polygon vertex in $O(m)$ total time. Therefore, we can build the visibility graph of the triangulated simple polygon P in $O(m)$ time.*

PROOF. As the source of the shortest-path map moves around the perimeter of the polygon, each directed polygon or visibility edge is added to or deleted from the current shortest-path map at most twice. (This follows from Lemma 3.) The extension edge associated with the directed edge, if any, is added and deleted along with it. There are $2n+2m$ directed edges, and hence by Theorem 1 or the analogous claim for the depth-first search algorithm, the $n-1$ applications of the transformation procedure take $O(m)$ time altogether. □

5. Conclusion and Open Problems. This paper has presented an algorithm to find the visibility graph of a triangulated simple polygon in $O(m)$ time, where m is the number of edges in the graph. The algorithm works by computing the shortest-path map from each vertex of the polygon, then reading off the visibility edges incident to the source of the shortest-path map. Rather than compute each shortest-path map separately, the algorithm finds one shortest-path map in linear time, then builds the others as modifications of the first one. In particular, the shortest-path map with source t is built by modifying the one whose source is the clockwise neighbor of t along the polygon boundary. This procedure is fast because of the similarity between two shortest-path maps whose sources are neighbors on the polygon.

Because it uses finger search trees to find the initial shortest-path map, our algorithm may be difficult to implement. However, we can avoid the use of finger search trees at the cost of slightly increased asymptotic complexity. We replace the finger search trees in the algorithm of Guibas *et al.* [GHLST] with linked lists; this gives an $O(n \log n)$ algorithm to find the first shortest-path map. (Note that we do not need a triangulation of P as input. We can afford to triangulate P using the $O(n \log n)$ algorithm of Garey *et al.* [GJPT].) We use the sweeping-path algorithm to transform shortest-path maps into each other. This yields an $O(m+n \log n)$ algorithm that uses only simple data structures.

The algorithms of Section 3 can be simplified if we assume that P is nondegenerate. If no three points of P are collinear, then any triangular shortest-path map region has at most five vertices on its boundary. In this case the sweeping-path algorithm can afford to visit all the vertices on the boundary of $P(\overline{st})$: the algorithm adds or deletes an edge at every such vertex. The depth-first search algorithm benefits from similar simplifications.

Even if degeneracies are allowed, an alternative definition of visibility can simplify our algorithms. If we define two points to be mutually visible when the segment connecting them does not intersect the exterior of P, the size of the visibility graph may increase. In a polygon with many collinear points, we may have $m = \Omega(n^2)$ under this definition and $m = O(n)$ under the standard definition. Under this broader definition of visibility, the sweeping-path algorithm will run in $O(m)$ time without using special cases to move p past the vertices on extension edges.

If we are given an untriangulated polygon as input, our algorithm takes $O(m+n \log\log n)$ time. The presence of the $O(n \log\log n)$ term in the running time suggests an open question: can we triangulate a simple polygon in $O(m)$ time? If we could do so, we would have an optimal visibility graph algorithm for simple polygons. (There are polygons with $m = O(n)$ for which the triangulation algorithm of Tarjan and Van Wyk takes $\Omega(n \log\log n)$ time.) This question fits into a framework of earlier work. Several authors have proposed triangulation algorithms that run in time $O(n \log k)$ for some parameter k related to the complexity of the polygon. For example, in the algorithm of Hertel and Mehlhorn, k is the number of reflex angles in the polygon [HeM]; in that of Chazelle and Incerpi, k is the "sinuosity" of the polygon [CI]. In some sense, we ask less of our triangulation algorithm than these authors. We want it to run in $O(m)$, but this can be much larger than $O(n \log\log n)$. When m is smaller than $O(n \log\log n)$, the polygon is narrow and few triangulations are possible. This may make it easier to find one of them.

Acknowledgments. I would like to thank Joseph O'Rourke for mentioning this problem to me, Leo Guibas for several helpful discussions, and Jack Snoeyink for his careful reading of an earlier version of this paper.

References

[AAGHI] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai: Visibility of disjoint polygons. *Algorithmica*, Vol. 1, No. 1 (1986), pp. 49–63.

[B] B. Baumgart: A polyhedral representation for computer vision. *Proceedings of the AFIPS National Computer Conference*, Anaheim, California, 1975, pp. 589–596.

[CG] B. Chazelle and L. Guibas: Visibility and intersection problems in plane geometry. *Proceedings of the ACM Symposium on Computational Geometry*, Baltimore, 1985, pp. 135–146.

[CI] B. Chazelle and J. Incerpi: Triangulation and shape complexity. *ACM Transactions on Graphics*, Vol. 3 (1984), pp. 135–152.

[GJPT] M. Garey, D. Johnson, F. Preparata, and R. Tarjan: Triangulating a simple polygon. *Information Processing Letters*, Vol. 7, No. 4 (1978), pp. 175–180.

[GM] S. K. Ghosh and D. M. Mount: An output sensitive algorithm for computing visibility graphs. *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, Los Angeles, 1987, pp. 11–19.

[GHLST] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan: Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, Vol. 2, No. 2 (1987), pp. 209–233.

[GMPR] L. Guibas, E. McCreight, M. Plass, and J. Roberts: A new representation for linear lists. *Proceedings of the Ninth ACM Symposium on Theory of Computing*, Boulder, Colorado, 1977, pp. 49–60.

[GS] L. Guibas and J. Stolfi: Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, Vol. 4, No. 2 (1985), pp. 74–123.

[HeM] S. Hertel and K. Mehlhorn: Fast triangulation of a simple polygon. *Proceedings of the Conference on Foundations of Computing Theory*, New York, 1983, pp. 207–218.

[HuM] S. Huddleston and K. Mehlhorn: A new data structure for representing sorted lists. *Acta Informatica*, Vol. 17 (1982), pp. 157–184.

[S] S. Suri: Private communication, 1986.

[TV] R. Tarjan and C. Van Wyk: An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, Vol. 17, No. 1 (1988), pp. 143–178.

[W] E. Welzl: Constructing the visibility graph for n line segments in $O(n^2)$ time. *Information Processing Letters*, Vol. 20 (1985), pp. 167–171.