8

VISIBILITY ALGORITHMS

8.1. INTRODUCTION

The notion of visibility leads to a number of algorithm questions independent of those motivated by art gallery problems. Although the structure of visibility graphs was investigated in the previous chapter, for example, we have yet to discuss the algorithmic construction of such graphs. Nor have we shown how to compute the portion of a polygon visible from an internal point. These and related questions will be addressed in this chapter.

The most fundamental problem is that just mentioned: given a point x in a polygon P, compute $V(x)$, the portion of P visible from x. $V(x)$ is called the *point visibility polygon* for x; it may be imagined as the region illuminated by a light bulb at x. It will be shown in the next section that $V(x)$ can be constructed in $O(n)$ time. Permitting holes in the polygon leads to $\Omega(n \log n)$ complexity (Section 8.5.1). In three dimensions, computation of $V(x)$ is the heavily studied "hidden surface removal" problem, which has recently been shown to have $\Theta(n^2)$ complexity (McKenna 1987).

Recall the definition of a kernel from Chapter 4: the *kernel* of a polygon P is the set of all points that can see the entire interior of P. Polygons with a non-null kernel are called *stars*.[1] Lee and Preparata showed that the kernel of a polygon can be computed in $O(n)$ time, which incidentally yields an algorithm for detecting whether a polygon is a star in linear time (Lee and Preparata 1979). We will not present their algorithm, but will use the idea of a kernel to introduce edge visibility.

Avis and Toussaint introduced and studied three different notions of edge visibility, extending the point visibility concept (Avis and Toussaint 1981b). Let P be a polygon and e an edge of P.

(1) P is *completely visible* from e if e is covered by the kernel of P: thus every point of P is visible to every point of e.
(2) P is *strongly visible* from e if e intersects the kernel of P: thus there is at least one point of e that can see all of P.

1. Such polygons are often called "star-shaped."

(3) *P* is *weakly visible* from *e* if every point of *P* is visible to some point of *P*.

Note that in the case of weak visibility, *e* does not have to intersect the kernel, and in fact *P* does not have to be a star to be weakly visible from an edge. An equivalent formulation is that a polygon is weakly visible from *e* if it would be entirely illuminated by a fluorescent light bulb whose extent matched *e*.

Avis and Toussaint addressed the question of detecting whether a polygon is visible from a given edge. This question is solved by Lee and Preparata's kernel algorithm for both complete and strong visibility, but not for weak visibility. They presented an $O(n)$ algorithm for detecting if *P* is weakly visible from *e* in Avis and Toussaint (1981b). We will not present their algorithm, but will make use of their definitions and theorems. The concept of weak visibility has proven to be the most fruitful of the three definitions, and henceforth the unqualified term "edge visibility" will refer to weak visibility.

A problem raised but not solved in Avis and Toussaint (1981b) is that of computing the edge visibility polygon $V(e)$ from an edge *e* of a polygon *P*: the portion of *P* illuminated by a light along *e*. For six years the fastest algorithms required $O(n \log n)$ time, but no lower bound larger than the trivial $\Omega(n)$ was known. Just recently an $O(n \log \log n)$ algorithm has been found, based on the Tarjan-Van Wyk triangulation algorithm (Section 1.3.2). We present an $O(n \log n)$ algorithm in Section 8.3, and sketch the new algorithm in Section 8.7. In Section 8.6 we will show that permitting holes in the polygon leads to a surprising jump in complexity to $\Omega(n^4)$.

A number of related visibility questions are surveyed in Section 8.7.

Finally, a remark on the style of algorithm presentation. Visibility algorithms tend to be complicated, involving, for example, delicate stack manipulations. It is not my intent to present these algorithms in the detail necessary for implementation; for that the reader is referred to the original papers. Rather I will attempt to convey the main ideas behind each algorithm while staying one step above the precise data structure manipulations.

8.2. POINT VISIBILITY POLYGON

The first linear algorithm for constructing the visibility polygon from a point inside a polygon was obtained by ElGindy and Avis in 1980 (ElGindy and Avis 1981). Prior to this, several supra-linear linear algorithms were published, and at least one suggested linear algorithm was shown not to work. ElGindy and Avis's algorithm requires three stacks, and is quite complicated. Later Lee proposed another linear algorithm that requires only one stack (Lee 1983). Most recently, Joe and Simpson have simplified the organization of Lee's algorithm (Joe and Simpson 1985), and it is their presentation that we follow here.

In order to achieve linear time, the vertices of the polygon cannot be sorted into a convenient organization, but rather must be processed in the order in which they appear on the boundary of the polygon. This order is inconvenient in that portions of the boundary not yet visited may obscure the otherwise visible portions of the boundary already visited. Thus the algorithm must be prepared to modify or abandon the structures it has constructed at any time.

Lee's algorithm accomplishes this with a single stack of vertices $S = s_0, s_1, \ldots, s_t$, where s_t is top of the stack. Let x be the point in the polygon from which visibility is being computed. Then the stack constitutes the vertices of $V(x)$ encountered so far *assuming the remaining portion of the boundary will not interfere.* Of course this assumption is in general not true, and as interference is detected, the stack is modified appropriately.

Let the vertices of the polygon be $v_0, v_1, \ldots, v_n = v_0$ in counterclockwise order. Place x at the origin and rotate and renumber so that v_0 is to the right of x on the horizontal line through x. For each vertex v_i of P, define its *angle about* x $\alpha(v_i)$ to be the polar angle of v_i with respect to x, including any "winding" about x. This may be defined formally as:

(1) $\alpha(v_0) = 0$,
(2) $\alpha(v_i) = \alpha(v_{i-1}) + \sigma \cdot \mathit{angle}(v_{i-1}xv_i)$,

where $\sigma = +1$ if $xv_{i-1}v_i$ is a left turn, $\sigma = -1$ if a right turn, and $\sigma = 0$ if no turn. Thus if $\alpha(v_i) > 2\pi$, the boundary has "wound around" x from v_0 to v_i. It is clear that only vertices v with $0 \le \alpha(v) \le 2\pi$ are candidates for visibility from x.

The algorithm consists of three procedures: *Push, Pop,* and *Wait. Push* adds a new visible vertex to the top of the stack. *Pop* deletes one or more vertices from the stack when interference is detected. And *Wait* traverses a portion of the boundary known to be invisible, waiting for it to emerge back "into the light." With each call to *Wait* is associated a *window* W, which is a subsegment of the ray from x through s_t, one end of which is always s_t, and a direction (clockwise or counterclockwise) of passage. *Wait* traverses the boundary until it passes through W in the specified direction.

Each procedure is now described in more detail. Let v_iv_{i+1} be the current edge being processed, and let the stack be $s_0, \ldots, s_t$.

Push

When this procedure is entered, $\alpha(v_{i+1}) \ge \alpha(s_t)$ and $\alpha(v_{i+1}) \ge \alpha(v_i)$. Two cases are distinguished, depending on the relation of $\alpha(v_{i+1})$ with 2π.

Case a ($\alpha(v_{i+1}) \le 2\pi$). This is the "normal" case. v_{i+1} is pushed onto the stack, and i is incremented. The next action is determined by the new edge v_iv_{i+1}, as follows (note that now $s_t = v_i$). If $i = n$, the algorithm is finished. If $\alpha(v_{i+1}) \ge \alpha(v_i)$, then *Push* is called again. If $\alpha(v_i) > \alpha(v_{i+1})$, then if the boundary makes a left turn at v_i, v_iv_{i+1} obscures the stack (Fig. 8.1a) and *Pop* is called; and if the turn at v_i is a right turn, then the stack obscures v_{i+1} (Fig. 8.1b), and *Wait* is called with $W = s_t\infty$.

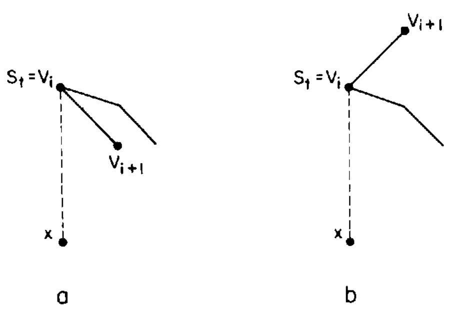

Fig. 8.1. If v_{i+1} obscures (a), *Pop* is called; if v_{i+1} is hidden (b), *Wait* is called.

Case b ($\alpha(v_{i+1}) > 2\pi$). Then the intersection of the ray xv_0 (which is at angle $0 \equiv 2\pi$) with v_iv_{i+1} is pushed on the stack, and *Wait* is called with $W = v_0s_t$.

Pop

The vertices of the stack are popped back to s_j, where s_j is the first stack vertex such that either

(a) $\alpha(s_{j+1}) \geq \alpha(v_{i+1}) > \alpha(s_j)$ (Fig. 8.2a), or
(b) $\alpha(s_{j+1}) = \alpha(s_j) \geq \alpha(v_{i+1})$, and y (defined in Fig. 8.2b) lies between s_j and s_{j+1}.

Case a. The stack top is set to point y in Fig. 8.2a, and i is incremented. The next action is determined by the new edge v_iv_{i+1}, similar to Case *a* of *Push*. If $i = n$, halt. If $\alpha(v_i) > \alpha(v_{i+1})$, then *Pop* is called again. If $\alpha(v_{i+1}) \geq \alpha(v_i)$, then if v_i is a right turn, call *Push*, and if a left turn, call *Wait* with $W = v_is_t$.

Case b. Ignoring the degenerate case when s_j, v_{i+1}, and s_{j+1} are collinear (see Joe and Simpson (1985)), *Wait* is called with $W = s_jy$, where y is as illustrated in Fig. 8.2b.

Wait

i is incremented until v_iv_{i+1} intersects W at point y from the correct direction. When that occurs, y is pushed on the stack, and either *Push* or

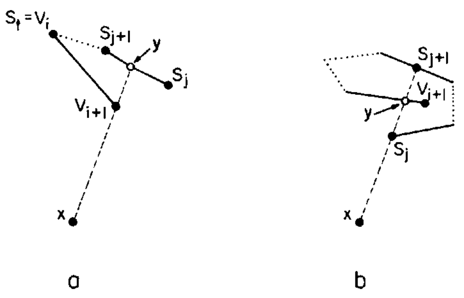

Fig. 8.2. Two *Pop* cases: v_{i+1} does (a) or does not (b) obscure s_{j+1}.

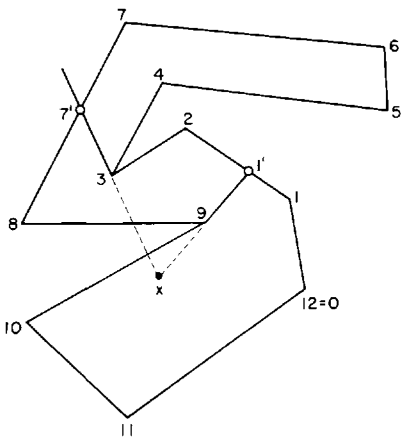

Fig. 8.3. A visibility polygon example: $V(x) = 0\,1\,1'\,9\,10\,11$.

Pop is called depending on whether $\alpha(v_{i+1}) \geq \alpha(v_i)$ or vice versa, respectively.

A simple example is shown in Fig. 8.3. *Push* advances to 3, when $S = 0\,1\,2\,3$. Since $\alpha(3) > \alpha(4)$ and 3 is a right turn, *Wait* is called with W as illustrated. *Wait* detects that 8 emerges through W, pushes 7′ on the stack, and calls *Push*. The stack becomes $0\,1\,2\,3\,7'\,8$ after 8 is pushed. Since $\alpha(8) > \alpha(9)$ and 8 is a left turn, *Pop* is called. All stack vertices down to 1 are deleted, and 1′ and 9 are pushed to make the stack $0\,1\,1'\,9$. Finally, *Push* advances until 0 is encountered again, when the stack is $0\,1\,1'\,9\,10\,11$, which is indeed $V(x)$. This example does not invoke the more subtle aspects of the algorithm, but illustrates the main ideas.

A proof of correctness requires more detailed code, and the interested reader is referred to the original papers (ElGindy and Avis 1981; Lee 1983; Joe and Simpson 1985). It should be apparent that the algorithm requires only linear time: each vertex is scanned just once, at most two vertices are pushed on the stack at each iteration, and popped vertices are never pushed again. Thus the time complexity is $O(n)$. Finally we note that the same basic algorithm can be used to construct the portion of the boundary of P seen from an exterior point x.

8.3. EDGE VISIBILITY POLYGON

In this section we discuss algorithms for computing the visibility polygon from an edge of a polygon. The generalization to polygons with holes will be considered in Section 8.6.

Let $V(e)$ be the portion of a polygon P visible from $e = (a, b)$. Of the three notions of edge visibility introduced in (Avis and Toussaint 1981b), only one, weak visibility leads to an interesting algorithm problem for constructing $V(e)$. It is easy to see that the region *completely* visible from e is just the intersection of $V(a)$ and $V(b)$. These point visibility polygons can be constructed in $O(n)$ time as showed in the previous section, and their intersection can be constructed easily in $O(n)$ time.[2] There is no unique region *strongly* visible from e; rather there are many regions strongly visible from an edge. But the construction of the region *weakly* visible from e, which we henceforth call $V(e)$, is a fascinating algorithm question that does not seem reducible to or from any other problem.

There have been three remarkably diverse algorithms published to date for constructing $V(e)$ in $O(n \log n)$ worst-case time complexity. And as this book was under revision, an $O(n \log \log n)$ algorithm was announced by Guibas *et al.* (1986). Their method will be sketched in Section 8.7. Here we will first outline each of the three published algorithms briefly before presenting a new fourth algorithm.

Independently and approximately simultaneously, ElGindy (1985), and Lee and Lin (1986a), proposed $O(n \log n)$ algorithms for computing $V(e)$. The two algorithms are completely different, and both are rather complicated. Lee and Lin's algorithm performs two scans of the polygon boundary in opposite directions, computing for each vertex the extreme points of e that can see it. The data gathered in the passes are then merged to form $V(e)$. Their algorithm maintains a separate stack for each vertex of the polygon. The reason for the $O(n \log n)$ complexity is that occasionally a binary search must be performed on a stack to search for a vertex with a particular property.

ElGindy's algorithm first decomposes the polygon into monotone pieces, using the $O(n \log n)$ algorithm of Lee and Preparata (see Section 1.3.2), and applies an edge visibility algorithm to each piece. Curiously he shows that a natural algorithm that achieves linear time for monotone polygons leads to a quadratic algorithm if applied to the monotone pieces. He uses an algorithm that requires $O(n \log n)$ even on monotone polygons, but which is better suited to merging the individual monotone results: it leads to an $O(n \log n)$ algorithm for computing $V(e)$ in a simple polygon.

A third algorithm was recently presented by Chazelle and Guibas (1985), and it is as different from the first two as they are from each other. The main novelty is that the calculations are carried out in a dual space using the "two-sided plane" introduced in Guibas *et al.* (1983). A convex partition of the rays comprising $V(e)$ in the dual space is constructed in $O(n \log n)$ time using a divide-and-conquer algorithm based on Chazelle's polygon cutting theorem (Chazelle 1982). Once this partition is available, $V(e)$ can be constructed in linear time. This approach is very general and solves several other visibility questions, to which we will return in Section 8.7.

Finally we come to the new fourth algorithm. It is a traditional plane sweep, based on several ideas in Lee and Lin (1986a) and ElGindy (1985).

2. I thank Subhash Suri for discussions on this point.

Let the edge e from which visibility is being computed be oriented horizontally. We concentrate initially on computing $V(e)$ above e; the portion of $V(e)$ below e (if any) is easily found with the point visibility algorithm applied to the two endpoints of e. The first step of the algorithm is to sort the vertices of the polygon from lowest y-coordinate to highest. This immediately pegs the complexity at $\Omega(n \log n)$. A horizontal sweep line H will be moved from e upwards. Let H intersect edges $e_1, e_2, \ldots$ left to right at a particular height. These edges are maintained in a data structure E that permits $O(\log n)$ queries, insertions, and deletions in the standard manner (see for example, Section 1.3.2). Assume for simplicity of exposition that no edge of the polygon aside from e is horizontal, so that the edges may be unambiguously classified as *left* or *right* edges, implying that the exterior of the polygon is to the left or right respectively. Clearly e_1 is a left edge, and they alternate left/right in sequence.

Certain pairs of left and right edges in E will be distinguished as bounding "visibility windows." A *visibility window* W is an interval of H bound by the left edge e_a on the left and the right edge e_b on the right, such that e_a and e_b are adjacent in E, and some portion of the interval between on H, specifically the interval $x_a x_b$, is visible to e. See Fig. 8.4. x_a may lay on e_a, or it may be that e_a is left of x_a (as in Fig. 8.5); and similarly for x_b. In general there will be several visibility windows $W_1, W_2, \ldots$ active on H at any one time. Each visibility window will have further data structures associated with it, which we now detail.

With each point x on H visible to e we can associate two line segments $L(x)$ and $R(x)$ that connect x to the leftmost and rightmost points of e that can see x. Construction of these lines in $O(\log n)$ time for any x is the key to the algorithm. Let $c[L(x)]$ be the point of contact between $L(x)$ and the polygon; in case there are several, select the one closest to x. Define $c[R(x)]$ similarly. Each window can be partitioned into intervals wherein the points of contact remain unchanged as x varies over the interval. The locations x where $c[L(x)]$ changes determine the left critical lines $L_0, L_1, \ldots, L_l$, and similarly there are right critical lines $R_0, R_1, \ldots, R_r$ for each window. For any window, the contact points for these lines form a convex chain, as illustrated in Fig. 8.5. As in that figure, the left critical lines intersect H in the left-to-right order $L_l, \ldots, L_1, L_0$, and the right lines in the order $R_0, R_1, \ldots, R_r$, where smaller indicies connect to lower points on the contact chains. It will always be the case that R_0 connects to x_a and L_0 connects to x_b. Points of H to the left of R_0 and to the right of L_0 are not visible to e. For each window, the critical lines and their contact points are maintained in two data structures L and R that permit $L(x)$ and $R(x)$ to be

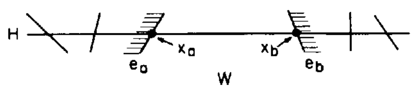

Fig. 8.4. A window W on the sweep line H; the shading represents the exterior of the polygon.

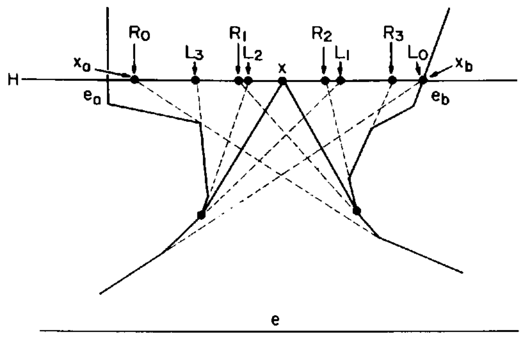

Fig. 8.5. The critical lines L_i and R_i, and the points of contact for x.

constructed in $O(\log n)$ time for any x in the window. As illustrated in Fig. 8.5, if x falls between L_i and L_{i+1}, then $c[L(x)] = c[L_i]$, and if x falls between R_i and R_{i+1}, then $c[R(x)] = c[R_i]$. Any standard dictionary data structure will suffice.

This completes the description of the data structures maintained as the sweep line advances. We now describe the actions taken as the line advances one step. Let H be the sweep line as it encounters the next vertex x. First x is located in the list of edges E in $O(\log n)$ time. If x is not interior to or on the boundary of any visibility window, the edges adjacent to x are inserted into and deleted from E in the standard manner in $O(\log n)$ time, and no further action is taken. If instead x lies in a window W, then three actions are taken: (1) visible segments in the window are output, (2) updates to the window due to the advance are made, and (3) updates to the window due to x are made. Each of these actions is described in detail below.

(1) Output of visible segments.

We will only describe the actions taken on the left boundary of the window; the right boundary is handled in the exact same manner. We first compute x_a, the leftmost visible point in W. The intersection of e_a, the left bounding edge of W, with H, y_a, is computed and located in the list of right critical edges R in $O(\log n)$ time. If y_a is visible, then $x_a = y_a$. Therefore, set x_a to be this point y_a if y_a is to the right or on R_0 (see Fig. 8.6); otherwise y_a is not visible and x_a is set to the intersection of R_0 with H. Let x_a' be the leftmost visible point of W when it was last updated, with

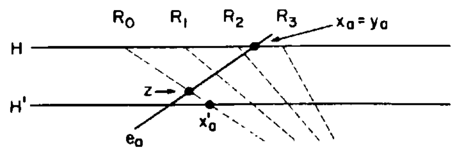

Fig. 8.6. The visible boundary segments $x_a' z$ and $z x_a$ are output.

the sweep line at H'. (This is not the immediately previous position of H in general, because each window is only updated when a vertex is encountered within it.) x_a' is the intersection of R_0 and H'. We now output the left boundary of the window from x_a' to x_a as visible. This boundary may consist of one or two segments:

(a) y_a is invisible, and so is strictly left of x_a. Then no portion of e_a between H' and H is visible, and both x_a' and x_a lie on R_0. Output one segment, $x_a'x_a$.
(b) y_a is visible, and so $x_a = y_a$ (Fig. 8.6). Then R_0 and e_a cross at a point z between H' and H; perhaps $z = x_a'$. Output two segments, $x_a'z$ and zx_a.

(2) Window updates due to advance.
Again we will only describe the updates related to the left bounding edge. Suppose $y_a = x_a$ is found to lie between R_i and R_{i+1}. The lines $R_0, R_1, \ldots, R_i$ are deleted from the data structure R. In Fig. 8.6, R_0, R_1, and R_2 are deleted. If any lines are deleted, then a new R_0 is created connecting x_a to $c[R_i]$. Similarly, x_a is located within L; if x lies between L_{j+1} and L_j, then $L_l, \ldots, L_{j+2}, L_{j+1}$ are deleted, and a new L_{j+1} is created (if any lines were deleted) connecting x_a to $c[L_j]$. If x_a is found to be to the right of L_0, or $R_0 = L_0$, then the window is closed.

(3) Window updates due to x.
We finally come to the processing that is dependent on the vertex x hit by H and its local neighborhood. Two cases are distinguished.

Case a ($x = y_a$ is the upper endpoint of e_a.). The edge distinguished as the left boundary of W must change. Let e' be the other edge incident to x. If e' is a left edge, then $e_a \leftarrow e'$; if e' is a right edge, then e_a is set to the first left edge to the left of x in E. Similar processing occurs when x is the upper endpoint of the right boundary.

Case b (The two edges e' and e'' adjacent to x to the left and right respectively both lie above H.). W splits into two windows W' bound by e_a and e', and W'' bound by e'' and e_b. Let x be located between R_i and R_{i+1}, and between L_{j+1} and L_j. The data structures R and L are split between the two windows, with W' receiving $R_0, \ldots, R_i$ and W'' receiving $R_{i+1}, \ldots, R_r$, and W' receiving $L_l, \ldots, L_{j+1}$ and W'' receiving $L_j, \ldots, L_0$. Note that this means that the top of the left convex chain and the bottom of the right convex chain become associated with W', and vice versa for W''. Finally, $L(x)$ and $R(x)$ are added to both W' and W''. For example, if x in Fig. 8.5 falls under Case b, then $i = 1$ and $j = 1$, and W' receives L_3, L_2, and $L(x)$, and R_0, R_1, and $R(x)$, and W'' receives $L(x)$, L_1, and L_2, and $R(x)$, R_2, and R_3.

This completes the description of the processing that occurs during each advance of H. The data structures are initialized with H collinear with e. E is initialized to contain every edge intersected by the initial position of H.

Let a and b be the left and right endpoints of e, and let e_a and e_b the edges in E closest to a and b respectively (a may be a lower endpoint of e_a, and similarly for b). Then there is one window initially, bound by e_a and e_b, which intersect H at x_a and x_b. Both L and R consist of two lines each: $L_0 = ax_b$, $L_1 = ax_a$; $R_0 = bx_a$, $R_1 = bx_b$.

A detailed proof of correctness would not be worthwhile in the absence of a more detailed description of the algorithm, which we have not provided. A few remarks about time complexity are in order, however. The sweep line advances exactly n times, once per vertex. At each advance, at most one window is updated. This is an important point, as it might seem natural to update all active windows with each advance. This, however, leads to a quadratic algorithm, and is not necessary: no visible segments can be lost by postponing window updating until a vertex is encountered within it. Each window update requires $O(\log n)$ time for data structure searches and updates, and constant processing to output the visible segments. Thus the total time complexity is $O(n \log n)$.

8.4. VISIBILITY GRAPH ALGORITHM

In this section we describe an $O(n^2)$ algorithm for constructing the visibility graph between the endpoints of a set of line segments. This is a very general problem, including, for example, construction of the visibility graph between vertices of a polygon with or without holes as special cases. Since so little is known about the structure of such graphs, however, the algorithm for line segments remains the fastest known algorithm for these special cases.

Perhaps the strongest motivation for the construction of visibility graphs is its application to the shortest-path problem. Lozano-Perez and Wesley showed that the shortest-path for a polygon amidst polygonal obstacles can be solved in $O(n^2)$ time using Dijkstra's shortest graph path algorithm applied to a certain visibility graph (Lozano-Perez and Wesley 1979). For several years the fastest algorithm known for constructing this visibility graph was $O(n^2 \log n)$; one such algorithm, for example, appeared in Lee's thesis (Lee 1978). Recently Welzl improved this to $O(n^2)$ (which is worst-case optimal) by exploiting an algorithm developed for constructing line arrangements.[3] This immediately gives an $O(n^2)$ algorithm for the shortest-path problem.

We will describe Welzl's algorithm here, taking time to explain the rudiments of the now considerable theory on line arrangements, which we will use again in Section 8.6.

Consider the set of three line segments shown in Fig. 8.7. The edges of the corresponding visibility graph G are drawn dashed in the figure. The

3. Several others discovered similar algorithms independently and slightly later; for example, Asano, Asano, Guibas, Hershberger, and Imai (1986).

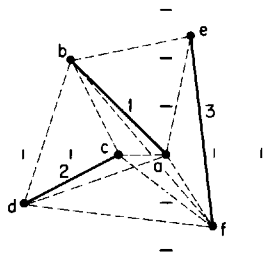

Fig. 8.7. A sample set of line segments. The origin is at a, and the unit hash marks on the (invisible) axes through a indicate the scale.

nodes of G are the endpoints of the line segments, and the arcs correspond to lines of visibility between endpoints. For the purposes of this section, we consider two points x and y visible to one another if the open segment (x, y) does not intersect any segment. This definition could be modified to permit "grazing contact" without altering the complexity of the algorithm. We first exhibit Lee's $O(n^2 \log n)$ algorithm for construction of G.[4]

The n endpoints determine $\binom{n}{2} = O(n^2)$ lines; in Fig. 8.7, $\binom{6}{2} = 15$ distinct slopes are determined. We will assume throughout the remainder of this section that all the slopes are distinct, as they are in this example. Sort these slopes from $-\infty$ to $+\infty$ in $O(n^2 \log n)$ time, and let $\alpha_1, \alpha_2, \ldots$ be the resulting sequence of sorted slopes. We will now show that G can be constructed from this list by an "angular sweep" in $O(n^2)$ additional time.

Let the line segments be labeled $s_1, s_2, \ldots$ in arbitrary order. For any direction α and any endpoint x, let $S_\alpha(x)$ be the segment first hit by a ray from x in direction α. If no segment is hit, define $S_\alpha(x) = s_0$, where s_0 is the "segment at infinity." For example, for $\alpha = 1$ (i.e., 45°), the endpoints in Fig. 8.7 have these values:

x	a	b	c	d	e	f
$S_\alpha(x)$	3	0	1	1	0	0

Let S_α be the function defined by $S_\alpha(x)$ for all x, that is, the vector shown in the previous table. The algorithm constructs $S_{\alpha_1}, S_{\alpha_2}, \ldots$ using the fact that each vector of this sequence differs very little from the one that precedes it.

In particular, suppose S_{α_i} has been constructed, and α_{i+1} is determined by the vertices a and b, with a of smaller X-coordinate than b. The algorithm advances to α_{i+1}, updating the vector and perhaps outputing an edge of the visibility graph. Let the ray from a through b hit $S_{\alpha_i}(a)$ at point c, and let $|xy|$ denote the distance between points x and y. Four cases are distinguished:

(a) a an b are endpoints of the same segment (Fig. 8.8a). Then $S_{\alpha_{i+1}} = S_{\alpha_i}$.

4. The presentation follows Welzl (1985).

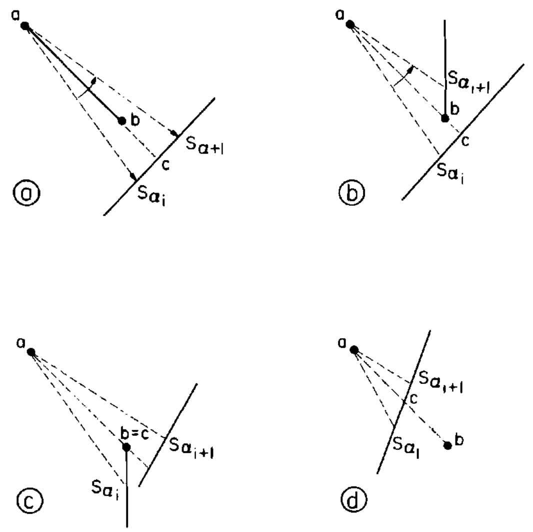

Fig. 8.8. Angular sweep transitions: the edge ab is output in (b) and (c) only.

(b) $|ab| < |ac|$ (Fig. 8.8b). Then $S_{\alpha_{i+1}} \leftarrow$ the segment containing b. Output edge ab.
(c) $b = c$ (Fig. 8.8c). Then $S_{\alpha_{i+1}}(a) \leftarrow S_{\alpha_i}(b)$. Output edge ab.
(d) $|ab| > |ac|$ (Fig. 8.8d). Then $S_{\alpha_{i+1}} = S_{\alpha_i}$.

It is clear that updating the vector requires only constant time per direction, as at most one element is altered, and its location can be accessed by pointers associated with each α_i. Thus a complete angular sweep takes $O(n^2)$ time, given an initial vector. This initial vector $S_{-\infty}$ can be constructed easily in $O(n \log n)$ time by a traditional plane sweep of a horizontal line. Thus G can be constructed in $O(n^2)$ given a sorting of the $O(n^2)$ directions. It remains an unsolved problem to obtain this sorting in better than $O(n^2 \log n)$ time, but Welzl showed that an *exact* sorting is not necessary: the angular sweep still works if the directions are only "topologically sorted" from the line arrangement. We now describe this clever idea.

The relevant line arrangement is dual to the set of endpoints. Let $p = (m, b)$ be a segment endpoint. Then the dual of p, T_p, is the line $y = mx + b$. Figure 8.9 shows the lines dual to the six endpoints of Fig. 8.7. The resulting structure is called an *arrangement of lines*. The lines dual to the two points $p_1 = (m_1, b_1)$ and $p_2 = (m_2, b_2)$, $y = m_1x + b_1$ and $y = m_2x + b_2$, intersect at

$$x' = \frac{b_2 - b_1}{m_1 - m_2} \quad \text{and} \quad y' = m_1\left(\frac{b_2 - b_1}{m_1 - m_2}\right) + b_1.$$

The line determined by p_1 and p_2 has slope $\dfrac{b_2 - b_1}{m_2 - m_1}$ and intercept

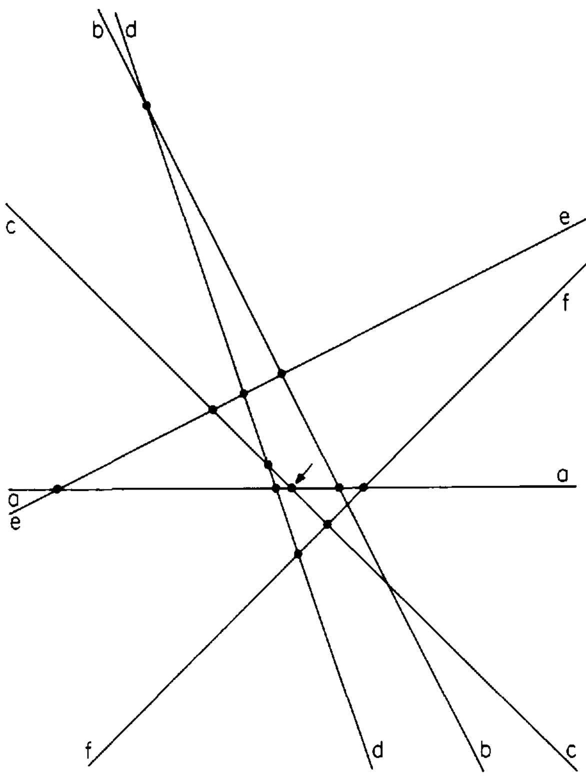

Fig. 8.9. The arrangement of lines dual to the vertices in Fig. 8.7. The arrow marks the origin of the coordinate system. The leftmost intersection (ae) has abscissa -5, and the rightmost (ef) has abscissa $+8$.

$-m_1\left(\frac{b_2-b_1}{m_2-m_1}\right)+b_1$. Thus the point of intersection (m, b) between T_{p_1} and T_{p_2} corresponds to the line $y=-mx+b$ passing through p_1 and p_2. If we imagine the line arrangement drawn in a space whose axes represent slope and intercept, then each intersection point in the arrangement corresponds to a direction determined by two endpoints, and the negative of the abscissa of an intersection point is the slope of the direction. Thus the ef intersection in Fig. 8.9 has abscissa 8, and the line determined by e and f in Fig. 8.7 has slope -8.

It should now be clear that a sorting of the intersection points in the arrangement from right to left corresponds directly to a sorting of the slopes determined by the endpoints, from smallest to largest. This is illustrated in Fig. 8.10, where all the distinct slopes derived from the point set of Fig. 8.7 are shown labeled with the points that determine them. Comparing with Fig. 8.9, we see that the order is preciely the right-to-left order of the intersection points in the arrangement.

It has been shown that the complete structure of a line arrangement of n

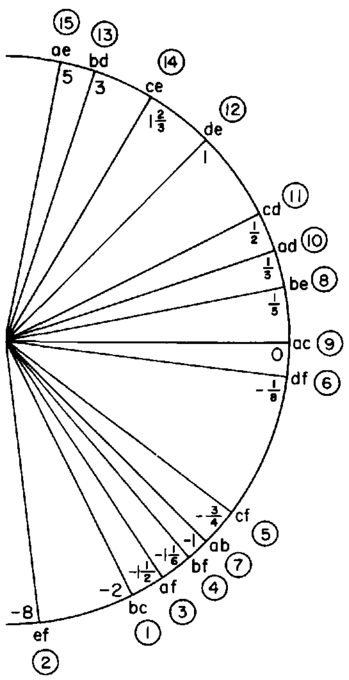

Fig. 8.10. The slopes of the intersection points in Fig. 8.9, labeled by the two lines meeting at that point, and by the slope. The circled numbers represent a topological sort.

lines that is, the incidence relations between all the vertices, edges, and faces determined by the lines, can be constructed in $O(n^2)$ time (Edelsbrunner *et al.* 1986; Chazelle, Guibas, and Lee 1985). This is a fundamental result which we will use but not prove. The correspondence between the vertices of a dual arrangement and the slopes of the directions determined by point pairs gives a great deal of structure to these slopes, but does not seem to lead to a sorting of them in $O(n^2)$ time. However, because the graph structure of the arrangement is available in $O(n^2)$ time, we can obtain a "topological sorting" of the intersection points quickly.

Define a directed graph D on the intersection points of the line arrangement as follows: there is a directed arc from vertex v to vertex u iff v is to the right of u, and u and v are connected by an edge of the arrangement. Figure 8.11 shows D for the arrangement in Fig. 8.9. A *topological sort* of a directed graph is an assignment of integers to the nodes such that the number assigned to a node is greater than all the numbers assigned to the nodes that connect to it with a directed arc. One topological sort (usually there are several) is indicated in Fig. 8.11. It is easy to perform a topological sort in time proportional to the size of the graph via a depth-first search (Aho *et al.* 1983); in our case the graph is of size $O(n^2)$. The labels assigned in Fig. 8.11 are also shown in Fig. 8.10, making it evident that a topological sort of the arrangement vertices does not

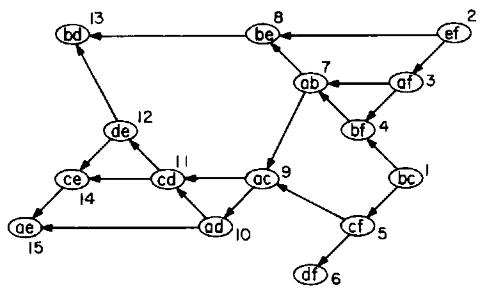

Fig. 8.11. The directed graph associated with the arrangement in Fig. 8.9, and a topological sort.

necessarily correspond to a sorting of the slopes. What Welzl proved, however, is that if the angular sweep algorithm is executed on the slopes organized by any topological sort, it will work just as well as it does with the slopes numerically sorted.

The reason is as follows. Consider all the intersection points on one line of the arrangement. For example, the line T_d dual to point d in Fig. 8.9 is intersected by the lines dual to points f, a, c, e, and b in that order from right to left. The sequence of these intersection points represents a sorting of the directions through d—an angular sweep centered on d. And notice that these intersection points must be sorted properly by the topological sort, since they all lie on a common line of the arrangement: the intersections with T_d are assigned the labels 6, 10, 11, 12, and 13 in Fig. 8.11. As long as all the directions through a common point x are processed in the order of their sorting about x, the angular sweep described previously will produce the correct result, because all of the relevant transitions in the $S_\alpha(x)$ function will be encountered in their correct order. Case (c) in Fig. 8.8 is critical: note that for the update from $S_{\alpha_i}(a)$ to $S_{\alpha_{i+1}}(a)$ to be correct, the value of $S_{\alpha_i}(b)$ must be known. But since the directions through b will be processed in the correct order, $S_{\alpha_i}(b)$ must be correct by the time the direction determined by a and b is considered. Table 8.1 shows the sequence of S_α vectors for our running example when the directions are processed in the topological sort order. Note that all the visibility edges are correctly output in one pass over the directions.

To summarize, Welzl's algorithm consists of the following steps:

(1) Construct the arrangement of lines dual to the endpoints of the line segments.
(2) Perform a topological sort of the vertices of the arrangement.
(3) Perform an angular sweep over the directions in the order given by the topological sort, updating the S_α function at each step, and outputing the edges of the visibility graph.

Each step can be accomplished in $O(n^2)$ time, thus yielding an algorithm for

Table 8.1. Each row shows an endpoint pair determining a direction α, and the S_α vector *after* sweeping past α. S_α elements in italics are the ones modified (or not modified) at direction α. Endpoint pairs shown in italics are output as edges of the visibility graph.

α	*a*	*b*	*c*	*d*	*e*	*f*
	0	2	0	0	0	0
bc	0	*0*	0	0	0	0
ef	0	0	0	0	*0*	0
af	*3*	0	0	0	0	0
bf	3	*3*	0	0	0	0
cf	3	3	*3*	0	0	0
df	3	3	3	*3*	0	0
ab	3	*3*	3	3	0	0
be	3	*0*	3	3	0	0
ac	3	0	*1*	3	0	0
ad	3	0	1	*1*	0	0
cd	3	0	1	*1*	0	0
de	3	0	1	*1*	0	0
bd	3	0	1	*0*	0	0
ce	3	0	*1*	0	0	0
ae	*0*	0	1	0	0	0

constructing the visibility graph in $O(n^2)$ time and space. That this is worst-case optimal follows from the fact that the visibility graph may have $\Omega(n^2)$ edges, for example, when each segment has length zero and no three endpoints are collinear.

It remains an open problem to construct a visibility graph in time proportional to its size, which can be $O(n)$ in special cases. The most recent advance in this direction has been made by Suri, who found an algorithm for constructing the vertex visibility graph of a polygon in time $O(k \log n)$, where k is the number of edges in the graph, using results from Chazelle and Guibas (1985).

8.5. POINT VISIBILITY REGION

In this section we extend the problem of computing a point visibility polygon $V(x)$, considered in Section 8.1, to an environment more general than the interior of a polygon: one consisting of n (perhaps disconnected) line segments. This includes polygons with holes as a special case. Because the resulting object $V(x)$ is not necessarily a polygon (it may be unbounded), we call it the point visibility *region.*

As might be expected, a linear algorithm is no longer possible in this more general case. We first establish that $\Omega(n \log n)$ is a lower bound, and then describe an algorithm that achieves this bound.

8.5.1. Lower Bound

We prove an $\Omega(n \log n)$ lower bound on the computation of point visibility inside a polygon with holes by reduction from the problem of sorting n integers, $(x_1, x_2, \ldots, x_n)$. Let $x_{\max}$ and $x_{\min}$ be the largest and the smallest numbers among $x_1, x_2, \ldots, x_n$, and let $\Delta = x_{\max} - x_{\min}$. Create an instance of the point visibility problem as follows.

The outermost polygon is a rectangle whose vertices are located at $(x_{\min} - 1, -\Delta/2)$, $(x_{\max} + 1, -\Delta/2)$, $(x_{\max} + 1, \Delta/2)$ and $(x_{\min} - 1, \Delta/2)$. With each number x_i, $1 \le i \le n$, associate a rectangular hole with vertices $(x_i - \varepsilon, -\varepsilon)$, $(x_i + \varepsilon, -\varepsilon)$, $(x_i + \varepsilon, \varepsilon)$, and $(x_i - \varepsilon, \varepsilon)$, where $\varepsilon = 0.1$, for example. Figure 8.12 illustrates the construction for $n = 4$. The point from which the visibility polygon is to be computed is set to be the lower left corner of the outer rectangle: $x = (x_{\min} - 1, -\Delta/2)$. It can be easily seen that the lower left corner of each hole occurs at every fifth vertex of the boundary of $V(x)$ in order of increasing values of x_i's. It is therefore easy to extract the sorted order of the x_i's from an algorithm that outputs the boundary of $V(x)$ as a list of vertices. Since sorting n integers is known to require $\Omega(n \log n)$ time in the general algebraic decision tree model, any such algorithm must spend $\Omega(n \log n)$ time in the worst-case.

We now exhibit a simple "angular sweep" algorithm that achieves this lower bound.

8.5.2. Algorithm

Let S be the set of line segments, assumed to intersect only at their endpoints, and let $P = (p_1, p_2, \ldots, p_n)$ be the set of endpoints of the segments of S. Assume without loss of generality that the given point x is the origin of our coordinate system and the set of points $P \cup \{x\}$ is in general position. Let D be the sequence of n sorted directions determined by x and the endpoints in P. We assume that the ray emanating from x along the X-axis has zero slope and the remaining slopes are measured counterclockwise about x.

The basic idea behind the algorithm is as follows. Let d be any ray. Let $s_1, s_2, \ldots, s_k$ be the sequence of segments of S that intersect d,

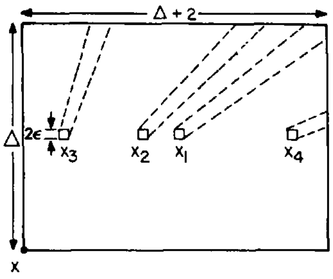

Fig. 8.12. Construction for the point visibility lower bound.

respectively, at $z_1, z_2, \ldots, z_k$ such that the segments s_1 through s_k are sorted by the rule $(s_i < s_j)$ iff $(|xz_i| < |xz_j|)$, $1 \leq i, j \leq k$, where $|xz_i|$ denotes the distance from x to z_i. Clearly, s_1 is on the boundary of $V(x)$. The algorithm rotates the ray around x and outputs the sequence of segments that intersect d first. The algorithm is, roughly speaking, an angular plane sweep, and may be described as follows.

Maintain a balanced binary tree T whose leaves are the segments that intersect the ray in the current direction. These segments are sorted by the rule described previously. The current direction is set to slope zero at the start of the algorithm, and then at each step advanced to the head of D, which is organized as a standard queue. The head of D is deleted at each step, and correspondingly a segment is either inserted or deleted from T. An interior node s of T stores the indices of the leftmost and the rightmost segments in the subtree rooted at s. Since the line segments do not cross, information stored with a leaf or an interior node does not change as the ray moves between two consecutive directions in D. For each direction in D, a segment is either inserted or deleted from T. Using the information stored with each node this segment can be inserted or deleted in $O(\log n)$ standard dictionary operations. T, therefore, can be arranged as a standard priority queue that permits the operations *insert, delete,* and *MIN* in logarithmic time per operation.

The correctness of the algorithm is straightforward. The time complexity can be established as follows. The sorted list of slopes, D, can be obtained in $O(n \log n)$ time. Initial construction of T can be accomplished in $O(n \log n)$ time since any ray d intersects $O(n)$ segments. At each step either a segment is added or deleted from T. Since a segment is added and deleted exactly once, and each deletion or insertion can be accomplished in $O(\log n)$ time, the algorithm runs in $O(n \log n)$ time, which is worst-case optimal.

8.6. EDGE VISIBILITY REGION

We generalize in this section the problem of computing the edge visibility polygon $V(e)$ to the general environment of a collection of line segment obstacles. In this environment, $V(e)$ may be unbounded, and it may have holes, so the term "region" is appropriate. Although it is not surprising that this problem has greater time complexity than the polygon case considered in Section 8.3, the magnitude of the complexity is perhaps unexpected: $\Omega(n^4)$. We first establish this lower bound before presenting an algorithm that achieves it.

8.6.1. Lower Bound

The bound is achieved by an example in which $V(e)$ has $\Omega(n^4)$ vertices on its boundary. This yields a lower bound on any algorithm that explicitly

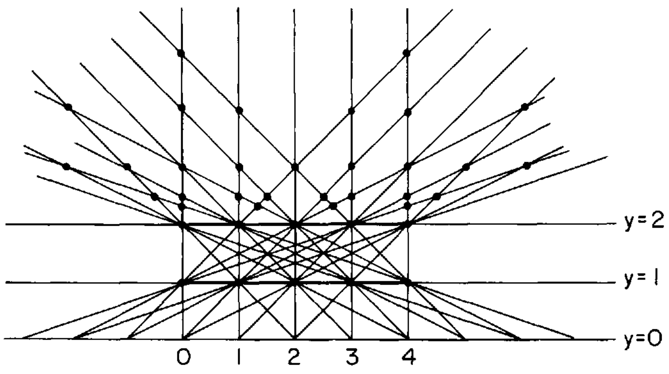

Fig. 8.13. Five gaps on two parallel lines ($y = 1$ and $y = 2$) above e ($y = 0$) produce 29 distinct intersections above the top line; in general, n gaps produce $\Omega(n^4)$ intersections.

constructs the boundary. The main idea of the example is as follows. Let the "luminescent" edge e be horizontal. Place n closely spaced line segments immediately above and parallel to e. The gaps between these segments permit $\Theta(n)$ cones of light to emerge above them. Place a second row of segments above the first, again parallel to e. $\Theta(n^2)$ beams of light escape above this second row. These beams intersect $\Theta(n^4)$ times above the second row, creating a region $V(e)$ with $\Omega(n^4)$ vertices and edges. See Fig. 8.13. A formal specification of this example follows.

Let the segment e have coordinates $\{(-n - 1/2, 0), (2n + 1/2, 0)\}$ for its two endpoints. The first set of segments H lies on the line $y = 1$. Each segment h_i is an open segment from a_i to b_i, where $a_i = (i, 1)$ and $b_i = (i + 1, 1)$ for $0 \le i \le n - 1$. Finally, two more open segments h_{-1} and h_n with the coordinates $\{(-n, -1, 1), (0, 1)\}$ and $\{(n, 1), (2n + 1, 1)\}$, respectively, are included. An identical set of segments H' is constructed on the line $y = 2$. Finally, enclose this set of segments in a rectangular polygon P whose corners have the coordinates

$$\{(-n - 2, -1), (2n + 2, -1), (2n + 2, n + 2), (-n - 2, n + 2)\}.$$

Now let S be the union of H, H', P', and e. Let g_i (respectively, g_i') denote the point gap between two consecutive segments h_{i-1} and h_i (respectively, h_{i-1}' and h_i'). Let G and G' denote the set of gaps for H and H', respectively. It should be clear that every pair of gaps $g_i \in G$ and $g_j' \in G'$, $0 \le i, j \le n$, determines a maximal line segment with one endpoint on e and the other on a side of P, and which does not intersect any other segment. We will call such a maximal line segment a "ray" (in slight abuse of standard terminology). There are $\Omega(n^2)$ rays altogether. Figure 8.13 shows the construction for $n = 4$; the outer polygon P is not shown. Let P' be the intersection of the half space $y \ge 2$ and the region bounded by P. It is clear that the visibility from e within P' is restricted to rays only. Therefore, an intersection point of two rays in P' is a vertex on the boundary of $V(e)$. If we can show that the $\Omega(n^2)$ rays intersect in $\Omega(n^4)$ distinct points in P', the bound will follow immediately. This may seem obvious, but in fact the intersection counting argument is somewhat involved because many rays are parallel, and many intersection points have more than two rays passing through them. One can make an irregular arrangement to avoid parallel beams and multiple intersections, but this also requires considerable care (Suri and O'Rourke 1985). Here we opt for the regular arrangement and proceed with the counting argument.

Let p be a point of intersection above $y = 2$ of at least three rays. Then p is the apex of at least two triangles based on the bottom row, as illustrated in Fig. 8.14. Let b_1 and a_1 be the widths of the left triangle at the bottom and top rows, respectively, and let b_2 and a_2 be the corresponding widths for the triangle that includes the left triangle; again see Fig. 8.14. Then we must have $\frac{b_2}{a_2} = \frac{b_1}{a_1}$ or $b_2 = \frac{b_1 a_2}{a_1}$. Since a_1, a_2, b_1, b_2 are all integers, a_1 must divide $b_1 a_2$. Suppose first that a_1 and b_1 are relatively prime. Then a_1 must divide a_2, and the larger triangle's width is an integer multiple of the smaller's. Suppose second that a_1 and b_1 are not relatively prime. Let $a_1 = ca'_1$ and $b_1 = cb'_1$, with a'_1 and b'_1 relatively prime. Then $b_2 = \frac{b'_1 a_2}{a'_1}$, which implies that a'_1 divides a_2. Let $a_2 = da'_1$. Substitution yields $b_2 = db'_1$. Thus a_2 and b_2 are not relatively prime. Thus both triangle widths are integer multiples of smaller triangles of widths a'_1 and b'_1.

The conclusion of this analysis is that all multiple intersections can be obtained as "scale multiples" of a leftmost, thinnest triangle with relatively prime a and b widths: leftmost because we are treating the scaling as expanding towards the right, and thinnest in that a and b are relatively prime. Thus the number of distinct intersections is equal to the number of leftmost, thinnest triangles. We now proceed to count these triangles.

Let a triangle be determined by a left line through $(b_1, 1)$ and $(a_1, 2)$ on the bottom and top rows, and a right line through $(b_2, 1)$ and $(a_2, 2)$, and let $b = b_2 - b_1$ and $a = a_2 - a_1$ (note the notation here is different from above). Let n be the number of gaps in each row, numbered from 1 to n.

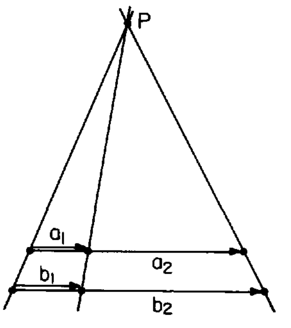

Fig. 8.14. Three lines coincident at one intersection point P determine two triangles, one included in the other.

The number of choices for each of these quantities is as follows:

b: b may take any value from 2 to n. $b = 1$ cannot result in an intersection above $y = 2$.

b_1: b_1 may range from 1 to $n - b$. We will partition this range from 1 to $\min(b, n - b)$, and the remainder.

b_2: b_2 is fixed at $b_1 + b$ once b is set.

a: If $a \geq b$, then the triangle does not result in an intersection point above $y = 2$; thus $a < b$. Moreover, a must be relatively prime to b, otherwise the triangle is not thinnest.

a_1: a_1 can range from 1 to $n - 1$ when $b_1 \leq b$, but only from 1 to a when $b_1 > b$, otherwise the triangle would not be leftmost. When $b_1 \leq b$ (and note that $\min(b, n - b) \leq b$), the situation is simpler; we will partition this range into two parts, from 1 to $n - b$, and the remainder.

a_2: $a_2 = a_1 + a$ cannot be greater than n, and since $a < b$, it must be less than $a_1 + b$. Within the range $a_1 = 1, \ldots, n - b$, the latter limit applies, and in the remainder the former limit applies.

To simplify the calculations, we will ignore the ranges of b_1 and a_1 that interact with the boundaries of the rows. Thus b_1 will range from 1 to $\min(b, n - b)$ and a_1 will range from 1 to $n - b$. Therefore, the quantity we obtain, $S(n)$, is a lower bound on the number of leftmost and thinnest triangles. Concatenating the four choices above yields the following formula:

$$S(n) = \sum_{b=2}^{n} \min(b, n - b)\phi(b)(n - b) \tag{1}$$

where $\phi(b)$ is the number of numbers less than b and relatively prime to b. We now show that this sum is $\Omega(n^4)$.

It is known that

$$\sum_{b=2}^{n} \phi(b) = \frac{3}{\pi^2} n^2(1 + o(1)) = \Omega(n^2) \tag{2}$$

See Grosswald (1966). The factors other than $\phi(b)$ in Equation (1) may be moved outside the summation by discarding the first and last quarter of the sum range. Equation (2) easily implies that $\sum_{b=n/4}^{3n/4} \phi(b) = \Omega(n^2)$. Using these summation limits, and replacing $\min(b, n - b)$ and $(n - b)$ in Equation (1) by their lower bounds of $n/4$ yields

$$S(n) > \left(\frac{n}{4}\right)\left(\frac{n}{4}\right) \sum_{b=n/4}^{3n/4} \phi(b) = \Omega(n^4).$$

Therefore, $S(n) = \Omega(n^4)$.

Table 8.2 shows the exact number of distinct intersections $I(n)$ for $n = 2, \ldots, 9$, where n is the number of gaps in each row. Figure 8.13 corresponds to the $n = 5$ entry.

Table 8.2

n	2	3	4	5	6	7	8	9
$I(n)$	0	2	11	29	69	125	224	361

It is necessary to modify the open segments used in the above construction to *closed segments*, to obtain a non-degenerate $V(e)$ with the same lower bound. This requires computing a sufficiently small rational number ε such that modifying the point gaps of our original constructions into ε-gaps, which enlarges the rays to beams, does not merge distinct intersection points. The calculation of epsilon is rather tedious (Suri and O'Rourke 1985); here we simply claim that $\varepsilon < 1/(cn^6)$ suffices, where c is a constant. The important point is that ε need not be exponentially small, which could make the input size larger than n under some models of computation.

8.6.2. Algorithm

We turn now to describing an $O(n^4)$ algorithm for constructing $V(e)$. The algorithm will only be sketched here; details may be found in Suri and O'Rourke (1985, 1986).

First observe that the boundary edges of $V(e)$ are either subsegments of the input segments S, or subsegments of lines through two endpoints in P such that the determined line intersects e.[5] We define a set E of line segments from which the boundary of $V(e)$ will be constructed as follows. First, henceforth consider e, the edge from which visibility is being computed, as a member of S. E consists of all line segments e_i such that:

(1) one endpoint p is in P;
(2) the other endpoint lies on a segment s_i in S, and the interior of e_i intersects no other segments of S;
(3) the line L containing e_i passes through another endpoint p_i in P, which may or may not lie on e_i; and
(4) the line L intersects e, and no other segments of S intersect L between e and p.

It should be clear that E may be constructed in $O(n^2)$ time by slight modification of Welzl's algorithm, described in Section 8.4. Whenever that algorithm outputs a visibility edge between p and p_i, the first segment s_i intersected by the extension of pp_i is available from the data structure. Supplementing the directions swept over with their negations insures that the extension in both directions will be considered. E can be easily constructed from this information.

Let $L(p) = (e_1, e_2, \ldots, e_k)$ be the list of edges of E with an endpoint at p, sorted angularly about p, where e_i terminates on s_i, and the line

5. Suri and O'Rourke (1985) for a formal proof of this claim.

containing e_i is determined by p and p_i, as in the definition above. It is somewhat less obvious that $L(p)$ can be obtained in $O(n)$ time for each $p \in P$ from the arrangement of lines used in Welzl's algorithm. Recall that the order of the intersections with the line dual to p in the arrangement corresponds to the directions determined by p sorted by slope. This basic observation can be used to extract $L(p)$ in linear time, as was shown in Asano *et al.* (1986). We will not prove this assertion here.

The algorithm performs an angular sweep about each $p \in P$ using $L(p)$, and outputs $O(n)$ triangular regions of visibility. The union of the resulting $O(n^2)$ triangles is then found in $O(n^4)$ time, and this constitutes $V(e)$.

Consider the sweep for a particular $p \in P$ from e_i to e_{i+1}. If p remains visible to e throughout the swept angle, then the triangular region between e_i and e_{i+1} is visible to e. There are four distinct cases, depending on the orientation of the segments whose endpoints are p_i and p_{i+1}. These are illustrated in Fig. 8.15, where the visible triangle to be output is shaded. The sweep is made through all of $L(p)$ for each $p \in P$. Note that since e is itself a member of S, triangles whose base is on e will also be output. The following lemma shows that the union of all these triangles is precisely $V(e)$.

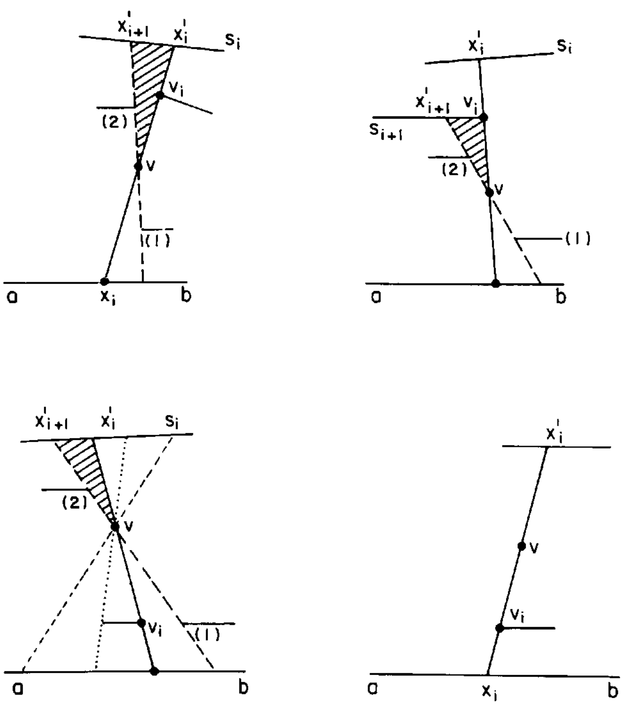

Fig. 8.15. Counterclockwise rotation about v may be blocked by a vertex at position (1) or (2). In (a)–(c), the shaded triangle is output; in (c), triangle $vs_ix'_i$ was output previously; in (d), no rotation is possible.

LEMMA 8.1. Let $T_i = \bigcup_j \Delta_{i_j}$, where Δ_{i_j} is a triangle rooted at $v_i \in P$ output by the just described algorithm. Then, $\bigcup_i T_i = V(ab)$.

Proof:

$\bigcup_i T_i \subseteq V(ab)$:

Each triangle output by the angular sweep is visible from e by construction.

$\bigcup_i T_i \supseteq V(ab)$:

We prove the claim by contradiction. Let $x \in V(ab)$ be any point such that $x \notin \bigcup_i T_i$. Let $y \in ab$ be any point that is visible from x. Imagine "swinging" the segment xy counterclockwise about x until it hits some vertex $z_1 \in P$. Let $y_1 \in ab$ and $x_1 \in s_x$ be the two points at which segment z_1x extended in both directions intersects the segments of S, where $s_x \in S$. Now, consider rotating the segment y_1x_1 clockwise about z_1 such that y_1x_1 maintains its contact with ab and s_x. Let z_2 be the first vertex of P contacted by this rotating segment x_1y_1. There are two cases to be considered, depending upon the relative positions of z_1 and z_2 (see Fig. 8.16). Notice

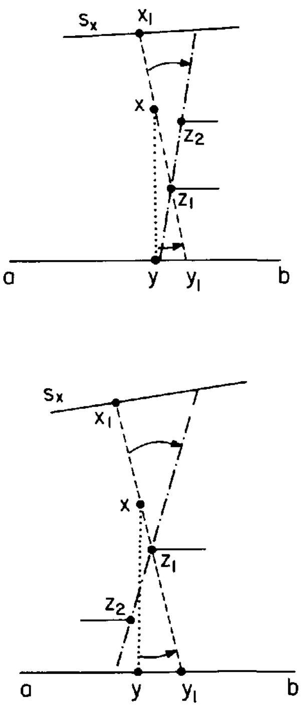

Fig. 8.16. x lies in a triangle rooted at z_1.

that the case $z_1 \in \{a, b\}$ is possible and does not require separate treatment. It is easily seen that in either case x lies in the triangle rooted at z_1 with one side collinear with the segment z_1z_2. But since this triangle belongs to $\bigcup_i T_i$ the assumption that $x \notin \bigcup_i T_i$ is contradicted. □

All that remains is the actual formation of the union of the $O(n^2)$ triangles. The problem of forming the union of polygons is very similar to a special case of hidden surface elimination: if the polygons are considered parallel to the xy-plane, we want the boundary of the view from $z = +\infty$. McKenna's hidden surface algorithm (McKenna 1987) requires $O(N^2)$ time for a scene with N vertices. Applying this algorithm with slight modification (see Suri and O'Rourke (1985)) to our triangles yields an $O(n^4)$ algorithm for construction of $V(e)$, which is worst-case optimal.

8.7. RECENT ALGORITHMS

Significant advances in visibility algorithms have been made as this book was being written. Here we mention three of the most important.

It was mentioned in Section 8.3 that the Chazelle–Guibas algorithm for constructing edge visibility polygons creates a data structure that can be used to solve other problems as well. Using this structure (and much else besides), they obtained the following strong result (Chazelle and Guibas 1985). There exists an $O(n)$ data structure for a polygon P that can be computed in time $O(n \log n)$, and which can answer queries of the following form in $O(\log n)$ time: given a point p in P and a direction u, find the first edge of P hit by a ray from p in the direction u. These so-called "bullet shooting" queries are the basis of Suri's output-size sensitive algorithm for construction of the vertex visibility graph of a polygon. In Guibas *et al.* (1986) the preprocessing time for Chazelle-Guibas algorithm is reduced to $O(n \log \log n)$.

Guibas *et al.* recently exploited the new $O(n \log \log n)$ triangulation algorithm to improve the asymptotic worst-case bounds for several visibility problems, most notably the problem of computing the edge visibility polygon (Guibas *et al.* 1986). First they showed how to compute the "shortest-path tree" from a vertex x of a polygon P: the union of all Euclidean shortest paths from x to every other vertex. This step depends heavily on the Tarjan-Van Wyk triangulation algorithm. They then prove that if $e = ab$ can see a portion of $e' = cd$, then the shortest paths from a to c and from b to d are both "outwardly convex," forming an hourglass shape (similar to that shown in Fig. 8.5). With this observation, they can construct $V(e)$ in a single boundary traversal of P using the shortest-path trees from a and from b. The result is an $O(n \log \log n)$ algorithm for computing $V(e)$.

The final result we will discuss here was invoked at the end of the previous section: the visibility region from a point in three dimensions can be computed in $O(n^2)$ time in an environment composed of polygons with a total of n vertices. This is the hidden surface elimination problem. The fastest algorithms developed until recently require $O(n^2 \log n)$ time in the worst-case (Sutherland *et al.* 1974), although they run much faster on the type of inputs encountered in practice. Recently McKenna used the $O(n^2)$ algorithm for constructing line arrangements to obtain an $O(n^2)$ worst-case optimal algorithm (McKenna 1987) for hidden surface removal.[6] This algorithm, however, is likely to be inferior to the standard graphics algorithms in practice. The major open problem in hidden surface algorithms is to find an output-size sensitive algorithm: one that runs in time $O(k \operatorname{polylog} n)$, where k is the number of line segments that are visible in the final scene. Such algorithms have only been achieved in special cases (Güting and Ottmann 1984).

6. This problem differs from that of hidden line elimination, which was solved in Devai (1986).