Discrete Comput Geom 4:551-581 (1989)

Discrete & Computational Geometry
© 1989 Springer-Verlag New York Inc.

Visibility and Intersection Problems in Plane Geometry*

Bernard Chazelle[1] and Leonidas J. Guibas[2]

[1] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA

[2] Department of Computer Science, Stanford University, Stanford, CA 94305, USA and
DEC Systems Research Center, Palo Alto, CA 94301, USA

Abstract. We develop new data structures for solving various visibility and intersection problems about a simple polygon P on n vertices. Among our results are a simple $O(n \log n)$-time algorithm for computing the illuminated subpolygon of P from a luminous side, and an $O(\log n)$-time algorithm for determining which side of P is first hit by a bullet fired from a point in a certain direction. The latter method requires preprocessing on P which takes time $O(n \log n)$ and space $O(n)$. The two main tools in attacking these problems are geometric duality on the two-sided plane and fractional cascading.

1. Introduction

Visibility and intersection problems are among the most fundamental topics in computational geometry. In this paper we investigate the following type of question: Given a simple polygon P and a pair (q, u) consisting of a point q and a direction u, imagine that we follow the path of a straight-line ray $\vec{r}$ from q in the direction u. We wish to know the first intersection (if any) of $\vec{r}$ with the boundary of P, or the intersections up to a limit point on the ray, or all the intersections. See Fig. 1.1. In posing this question we assume that P is fixed once and for all, so we are allowed to do preprocessing on it.

As a warm-up we begin our discussion in Section 2 by investigating the easier case where q is *confined to lie on a side of* P (the luminous edge). We give simple methods for:

1. Computing the illuminated subpolygon of P from the luminous edge in $O(n)$ space and $O(n \log n)$ time.

* Bernard Chazelle wishes to acknowledge the National Science Foundation for supporting this research in part under Grant CCR-8700917. A preliminary version of this paper was presented at the First Annual ACM Symposium on Computational Geometry, June 1985.

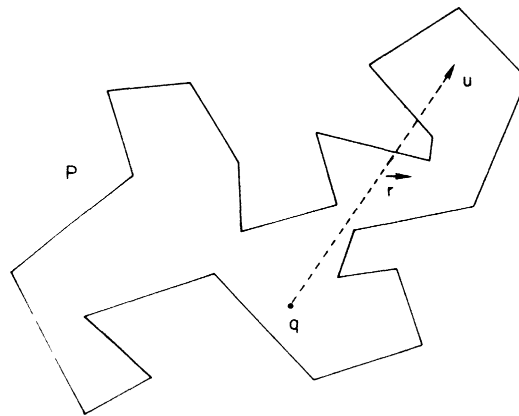

Fig. 1.1

2. Computing the first intersection of the boundary of P with a light-ray from the luminous edge in $O(\log n)$ time, given an $O(n)$-space data structure computable in $O(n \log n)$ time.
3. Computing all the points of the luminous edge that cast light on a particular point on the boundary of P in $O(\log n)$ time, using an $O(n)$-space data structure which is computable in $O(n \log n)$ time.

Problem 1 has been solved previously within these time bounds (see Section 2), but we believe that our solution is significantly simpler than the earlier ones. Our solutions to problems 2 and 3 improve on the previously known bounds. Another attractive aspect of our approach is the use of a common data structure for all three problems.

In Section 3 we solve the significantly more difficult *unrestricted version* of the ray-shooting problem. Our solution computes the first intersection in $O(\log n)$ time, using a structure that takes $O(n)$ space and $O(n \log n)$ preprocessing time. In the process of describing our solution we develop some machinery that we expect to be useful in other geometric problems as well.

Finally, in Section 4 we mention some extensions for computing more than the first intersection, or for dealing with objects more complex than a simple polygon. For instance, using our structures, we can compute all k intersections of a line segment and a simple polygon P in time $O((k+1)\log(n/(k+1)))$. This problem for the case of an infinite segment (a straight line) had been solved previously within the same time bound by Chazelle and Guibas [4].

One of our main tools for these problems is the *geometric duality* on the *two-sided plane* (2SP) introduced by the kinetic framework [10]. This is a classic duality, together with the convention that oriented lines having the origin to their left dualize to points on the top surface, while lines having the origin to their right dualize to points on the bottom side. We refer the reader to that paper for details. A crucial property for us will be that in the 2SP boundaries of convex polygons dualize to boundaries of convex polygons. The duality transform is

denoted by D, so for a point p, D_p will denote the dual oriented line, and for a line r, D_r will designate the dual point.

2. The Use of Duality in Visibility Problems

Let $P = p_1, p_2, \ldots, p_n$ be a *simple* polygon in the plane, and let e denote the edge p_1p_2. Imagine that e is a luminous neon bulb, while the other sides of P form opaque walls. This scenario raises a number of interesting questions (see Fig. 2.1):

P1: Compute the region of P illuminated by e.
P2: Given a light-ray $\vec{r}$ starting on e, determine which point on the boundary of P it illuminates (hits).
P3: Given a point q on the boundary of P, find the points of e, if any, that shine light on q.

As the above questions make clear, we assume that all points of e emit light-rays in all directions toward the interior of P. A minor technicality is to decide whether the endpoints of an edge should be luminous. For convenience we assume that they are not, which means that no edge can emit light behind. Should we like to relax this assumption we can always create edges of infinitesimal length at the endpoints, and, by this (rather devious) way, support vertex-visibility in addition to edge-visibility.

Our investigation of the three problems above is based on the observation that lines, and not points, are the primitive objects to consider in visibility questions. Since points are intuitively easier to grasp than lines, such questions are advantageously recast in dual space, where the roles of points and lines are interchanged. For problem P1, $O(n \log n)$ solutions were recently discovered independently by Lee and Lin [11] and El Gindy [7]. Both algorithms are fairly involved and

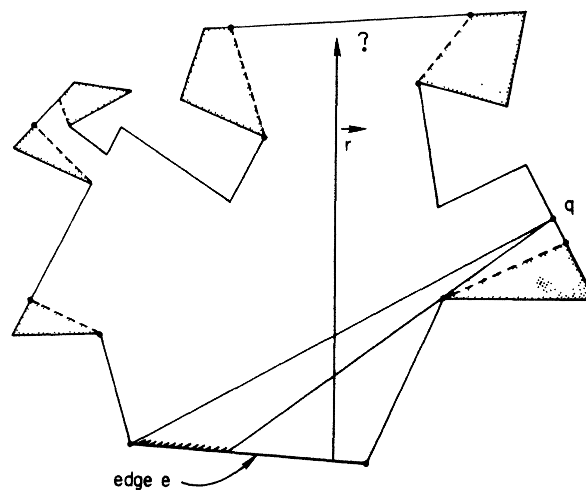

Fig. 2.1

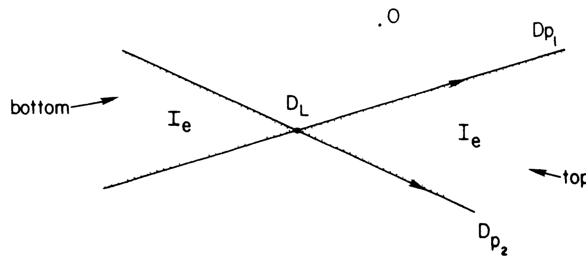

Fig. 2.2

entail substantial overhead, in part because of their reliance on dynamic search structures. As we will see, by using duality, we can obtain a single algorithm that not only produces the visibility region in $O(n \log n)$ time but also allows us to solve problems P2 and P3 in $O(n)$ space and $O(\log n)$ response time.

We will identify a light-ray $\vec{r}$ with its supporting oriented line r. Each such line dualizes to a point $p = D_r$ in the dual plane. It is easy to check that the collection of all oriented lines intersecting the segment e and (locally) directed toward the interior of P dualizes to a *double wedge* I_e. This double wedge is delimited by the lines D_{p_1} and D_{p_2} (recall that $e = p_1 p_2$) and is the one (of the two possible double wedges formed by D_{p_1} and D_{p_2}) *not* containing the origin. We let L denote the line supporting e. See Fig. 2.2. On the 2SP this double wedge is actually a convex polygon consisting of one wedge on the top and the opposite wedge on the bottom. Points on the top wedge correspond to rays passing to the left of the origin, while those on the bottom correspond to rays passing on the right.

Why is I_e convex? This is actually a consequence of some very general duality theorems. We can, however, see it directly by noting that if r_1 and r_2 are any two rays cutting e and directed into the same half-plane, then any ray r between r_1 and r_2 will also cut e and be directed toward the same half-plane. Here by "between" we mean that t passes through the intersection of r_1 and r_2 and is contained in the wedge cut by e which they form. See Fig. 2.3.

For a point $p \in I_e$ we define $f(p)$ to be the edge(s) of P hit by the light-ray D_p. The function f defines a partition $S(I_e)$ of the wedge I_e into the subregions where f is constant: two points p, q are in the same region of $S(I_e)$ if $f(p) = f(q)$.

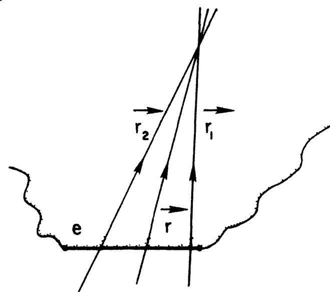

Fig. 2.3

Lemma 2.1. *Each region of the subdivision $S(I_e)$ is a convex polygon.*

Proof. Let p, q be two points in I_e with $f(p) = f(q) = \{g\}$, an edge of P. Let $A = e \cap D_p$, $B = e \cap D_q$, and $I = D_p \cap D_q$ (on the 2SP the intersection I is always well defined). It is geometrically obvious that any light-ray passing through I and crossing the segment AB will hit the edge g first. In dual terms, any convex combination $r = \lambda p + (1-\lambda)q$, $\lambda \in [0, 1]$, of p and q satisfies $f(r) = \{g\}$. Thus f defines at most $n-1$ two-dimensional convex regions which together subdivide the convex wedge I_e. It follows that each region must be a convex polygon (in the 2SP sense, of course). □

2.1. *Computing the Subdivision $S(I_e)$*

As we will see below, the subdivision $S(I_e)$ provides the essential information we need to know in order to solve problems P1-P3. In this section we concentrate on how to compute it efficiently. We use a divide-and-conquer technique based on the *polygon-cutting theorem* of Chazelle [1]. This theorem states that, with $O(n \log n)$ preprocessing, it is possible to determine two vertices p_i and p_j of P such that the diagonal $p_i p_j$ lies fully in P and subdivides P into two simple polygons each with at most $\lceil 2n/3 \rceil + 1$ vertices. This assumes that P has more than three vertices. Once the preprocessing is complete, the cost of computing the separating diagonal is linear, and it remains so as we apply the subdivision recursively to the subpolygons created. For example, this preprocessing could be the computation of a triangulation of P. Then each separating diagonal can be found by searching for the centroid of a free tree, which takes linear time. See [1] for details.

If P is a triangle, $S(I_e)$ is trivially computed in constant time. If P has more than three vertices, we determine the separating diagonal $p_i p_j$ and decompose P into two other polygons, P_1 and P_2. Without loss of generality we assume that e is an edge of P_1. We now compute recursively the subdivision S_1 associated with illuminating P_1 from e, and the subdivision S_2 associated with illuminating P_2 from $p_i p_j$. Let R be the region of S_1 which corresponds, via the function f, to the edge $p_i p_j$. In other words, R is the locus of the duals of all rays in P_1 emanating from e and hitting $p_i p_j$. A crucial observation is that

$$S(I_e) = S_1 \cup (R \cap S_2),$$

meaning that by clipping S_2 to within R and adding this refinement of R to S_1, we obtain $S(I_e)$. So, the desired subdivision $S(I_e)$ is obtained from S_1 by simply subdividing one of its regions. See Fig. 2.4.

We can carry out the above construction in $O(n)$ time. Why is that so? Let us start out by triangulating each region of S_2. Since these regions are convex, we can do this in time linear in the size of S_2, which is $O(n)$. Indeed, it trivially follows from Euler's relation and convexity that the description size of all the convex subdivisions S_i is $O(n)$. Next, we locate an arbitrary starting vertex of R

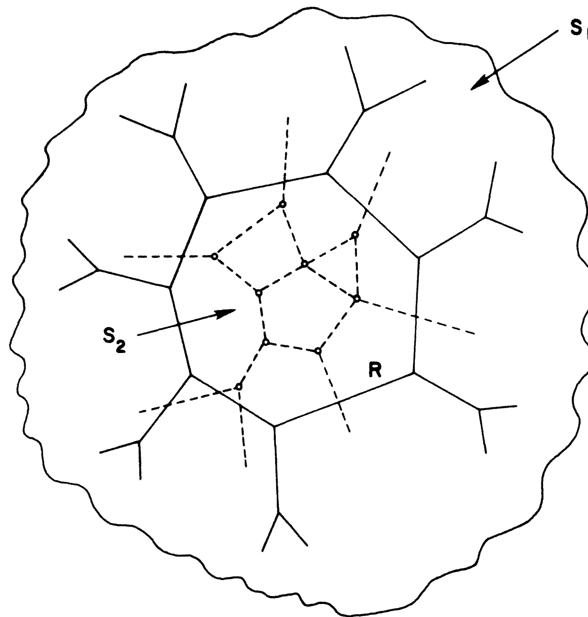

Fig. 2.4

in this triangulation and proceed to walk around R while computing all the intersection points of edges of S_2 and R. From there a simple depth-first search toward the interior of R allows us to construct $R \cap S_2$. By convexity, the boundary of R intersects no edge of S_2 more than twice, therefore the total complexity of the clipping operation is still $O(n)$, which establishes our claim.

Since the *merge* part of this divide-and-conquer algorithm is linear, the balanced decomposition provided by the polygon-cutting theorem leads to the following result:

Lemma 2.2. *It is possible to compute the convex subdivision $S(I_e)$ associated with illuminating the simple polygon P from edge e in $O(n \log n)$ time and $O(n)$ space.*

As we already observed, if F is the number of nonempty regions in the subdivision, its complete description size is $O(F)$. The regions correspond to the sides of P that receive some light from the luminous side (see also Section 2.2). The quantity $O(F)$ is also a bound on the size of the illuminated subpolygon of P from the luminous side e.

Another $O(n \log n)$ algorithm can be derived by starting out with a triangulation of P, and then considering the triangles one at a time, starting with the triangle containing the luminous edge. At each step we add a new triangle Δ having one side in common, say d, with a previously handled triangle. The rays reaching d (a diagonal of P) from e form a convex polygon V in the dual plane. They will now be subdivided into one or two groups, according to which other side of Δ they exit from. Dually, the polygon V will be cut into two subpolygons

by the line which is the dual of the new vertex of Δ. We can compute the final subdivision by continuing in this way and paying an $O(\log n)$ line/convex polygon intersection cost per triangle added.

2.2. Solving the Visibility Problems

In this subsection we show how problems P1–P3 can be solved using the subdivision $S(I_e)$.

Theorem 2.3. *Computing the subpolygon of P illuminated from a luminous edge can be done in $O(n \log n)$ time and $O(n)$ space.*

Proof. An edge g of P is, possibly partially, illuminated by e if and only if it is associated with a nonempty region R of $S(I_e)$. To determine which part of g is illuminated, we examine its associated region R. Let D_l be the dual point of the line l supporting side g (in some orientation). Any line passing through D_l and cutting through R dualizes to an illuminated point of g, and vice versa. It easily follows that the two lines passing through D_l and tangent to R are the duals of the two points A and B which delimit the visible part of g. (Why cannot D_l be in the interior of R?) Occasionally R will have an edge collinear with D_A (or D_B). In this case A (or B) will be an endpoint of g. Both A and B can be determined in time linear in the size of R and therefore this information can be collected for *all* edges of P in $O(n)$ time. To form the illuminated region of P from e we just join the endpoints of consecutive visible segments as they occur on the edges around P. See Fig. 2.1. □

Theorem 2.4. *Problem* P2 *can be solved in $O(\log n)$ time by using an $O(n)$ space data structure computable in $O(n \log n)$ time.*

Proof. As we have already seen, a light-ray from e becomes a point in I_e in the dual plane, and the region it lies in $S(I_e)$ directly tells us the edge of P that it hits. A straightforward calculation gives us the intersection point in constant time. Using an optimal point location algorithm, e.g., [6], the entire process can be carried out within the bounds stated by the theorem. □

Theorem 2.5. *Problem* P3 *can be solved in $O(\log n)$ time by using an $O(n)$ space data structure computable in $O(n \log n)$ time.*

Proof. A point q on an edge g of P is illuminated from e if and only if the line D_q intersects the region R associated with g. By duality, the line D_q passes through the point D_l, where l is the line supporting g. It is easy to check that the two intersection points A and B of D_q with the boundary of R are the duals of the two lines defining the visibility wedge from e to q. Consequently, intersecting these two lines with e will provide us with the subsegment of e illuminating q. Since the intersection of a convex polygon and a straight line can be computed in logarithmic time, the theorem follows. □

Natural extensions of the previous algorithms allow us to handle the case where the neon light does not coincide with an edge of P but corresponds to a line segment s lying inside the polygon. In this case we simply extend the segment s so as to split P into two subpolygons for which the previous techniques directly apply. (The fact that now only a contiguous portion of e is light-emitting introduces no difficulties.)

We may also wish to consider problem P2 for the case where the light-ray $\vec{r}$ comes toward P from infinity. For this situation we start by forming the convex hull of P. The difference between P and its convex hull is a collection of simple polygons ("bays"), each containing exactly one edge not in P. We preprocess these polygons for illumination from their special edge not in P. To discover now which side of P the ray $\vec{r}$ intersects, we first compute in $O(\log n)$ time the initial point of contact between $\vec{r}$ and the convex hull of P. If this point is on an edge of P we are done. Otherwise we are in an instance of problem P2 for one of the preprocessed polygonal bays. Since the total size of all these polygons is $O(n)$, we have shown that:

Theorem 2.6. *There exists an $O(n)$ space data structure representing a simple n-gon P that allows us to compute in $O(\log n)$ time the point on the boundary of P illuminated by a light-ray coming from infinity. This data structure can be built in $O(n \log n)$ time.*

3. The General Ray-Shooting Problem

In the previous section we discussed a number of instances of the *shooting problem*: preprocess a simple polygon P so that given a pair (q, u) consisting of a point q and a unit vector u, we can efficiently determine the first point on P to be hit by the ray from q in the direction u. This point, if it exists, is denoted by $\sigma^P(q, u)$. Problem P2 and its variants for which we have already presented solutions make essential use of the restriction that q is constrained to lie on some straight line. To solve an unrestricted instance of the shooting problem we can proceed as follows.

First we assume that q lies in the polygon P. If not, the same convex hull and polygonal bay trick we used in the previous section can be employed to reduce the problem to the case where q is interior to some polygon, or to an instance of the problem covered by Theorem 2.6.

As in Section 2, we apply the polygon-cutting theorem recursively to decompose P into a balanced tree T of polygons. Each node $v \in T$ is associated with a polygon $P(v)$ and a *cutting edge* $e(v)$, which is a diagonal of P separating the two polygons associated with the offspring of v. If v_1 and v_2 are the nodes corresponding to these two children, then we know that the sizes of $P(v_1)$ and $P(v_2)$ are roughly in a ratio between $\frac{1}{2}$ and 2, and each is no more than roughly two-thirds the size of $P(v)$. The leaves of T are associated with triangles, and the set of all leaves constitutes a triangulation of P. The following lemma, whose

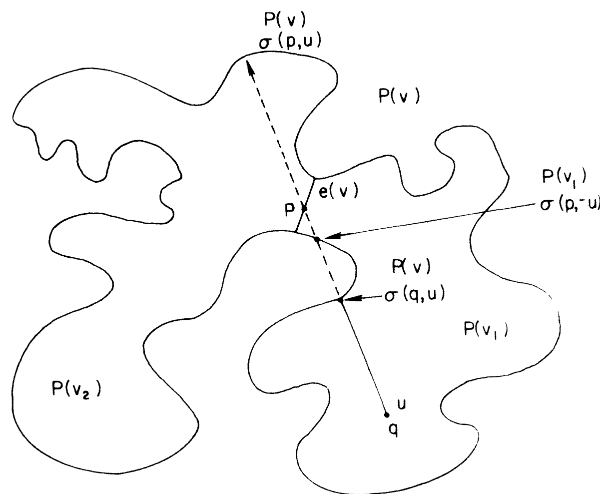

Fig. 3.1

proof is straightforward, can form the basis of a recursive algorithm for computing $\sigma^P(q, u)$. See Fig. 3.1.

Lemma 3.1. *Let v_1 and v_2 be the two children of v, and assume that q lies inside $P(v_1)$. The point $\sigma^P(q, u)$ lies in $P(v_2)$ if and only if the ray from q in the direction u intersects $e(v)$ at some point p and q lies on the segment $p\sigma^{P(v_1)}(p, -u)$. If this holds, then $\sigma^P(q, u) = \sigma^{P(v_2)}(p, u)$; else $\sigma^P(q, u) = \sigma^{P(v_1)}(q, u)$.*

This immediately suggests a recursive algorithm, *shoot*$(v, point, dir)$ for computing the point $\sigma^{P(v)}(point, dir)$. The algorithm is initially called with $v = x$, the root of T, $point = q$, and $dir = u$. If v is a leaf, compute $\sigma^{P(v)}(point, dir)$ directly, return the point found, and stop. Else let v_1, v_2 be the two children of v. By induction, *point* lies in $P(v)$. Perform a planar point location to decide if *point* lies in $P(v_1)$ or $P(v_2)$; without loss of generality let us assume that the former is the case. Let p denote the intersection of $e(v)$ and the ray from *point* in the direction *dir*. If p exists, then use the algorithm of problem P2 to compute $p' = \sigma^{P(v_1)}(p, -dir)$. If the segment pp' contains *point*, then call *shoot*(v_2, p, dir), or more simply, use the algorithm of problem P2 to complete the computation. If p does not exist, or *point* is outside pp', then call *shoot*$(v_1, point, dir)$.

The preprocessing required by the procedure *shoot* involves setting up the tree T, organizing each $P(v)$ for efficient planar point location, and applying the preprocessing of P2 to $P(v_1)$ and $P(v_2)$ with respect to $e(v)$, for each node $v \in T$ and children v_1, v_2. A straightforward analysis shows that all this needs $O(n \log n)$ space. Once these structures are in place, which with a bit of care takes $O(n \log n)$ time, the procedure *shoot* can be computed in time $O(\log^2 n)$. It corresponds to a walk down T, where at each step we may need to do a point location and solve an instance of P2.

This fairly naive implementation can be refined to save a factor of log n in both space and query time. These refinements require some novel structures, which we describe next.

3.1. Some Mathematical Preliminaries

The key observation we make use of is that at each node $v \in T$ the whole visibility structures of $P(v_1)$ and $P(v_2)$ with respect to $e(v)$ are not really needed. The useful information is whether we can shoot directly from $e(v)$ through $e(v_1)$ or $e(v_2)$ without hitting the intermediate portion of the polygon. This information can be encoded for all possible shooting rays with a simpler structure than an arbitrary planar subdivision. By formalizing this observation and using the *fractional cascading* technique of Chazelle and Guibas [4] we are able to avoid the cost of repeated planar point locations.

Our first task is to augment the tree T into a graph T^* by the addition of certain edges. This augmentation process was introduced in [3] and this is the place to which we refer the reader for details of this procedure. We add an edge between all pairs of nodes (v, w) such that $e(v)$ *is an edge of the boundary of* $P(w)$. Note that if this condition holds, then v is necessarily an ancestor in T of w. Figure 3.2 depicts (schematically) a polygon P hierarchically subdivided by the polygon-cutting theorem and the associated tree T. Figure 3.3 also shows the additional edges thrown in to form T^*. (Note that we have displayed only the top part of the tree, since leaves do not correspond to triangles.) If v_1 is a child of v, then $e(v)$ is clearly on the boundary of $P(v_1)$, so the edges of T trivially satisfy the condition to be in T^*.[1]

Let us assign levels to the tree T, where the root is given level 0 and a child is at a level one higher than its parent. Thus in our figures levels increase downward, and higher levels occur lower on the page (reader beware!).

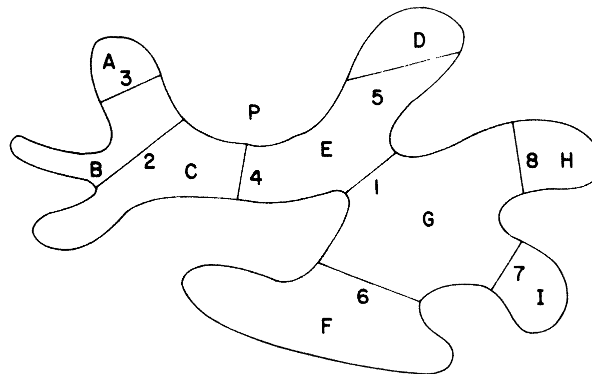

Fig. 3.2

[1] Unfortunately we might now use the word *edge* both for a diagonal of P and for an edge of T^*.

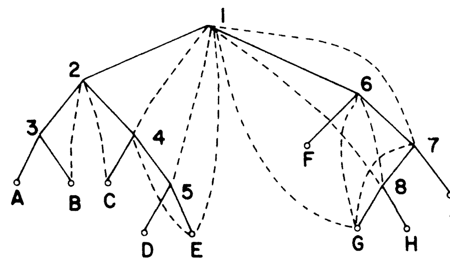

Fig. 3.3

Lemma 3.2. *A node v in T at level l has*

(i) *for each level l' smaller than l at most one T^* edge going to an ascendant node at level l', and*
(ii) *for each level l' larger than l at most two T^* edges going to descendant nodes at level l'.*

Therefore each node of T^* has degree at most $O(\log n)$, and all of T^* has size $O(n)$.

Proof. Parts (i) and (ii) are immediate consequences of the recursive nature of the splitting process. Adding these to the fact that the decomposition is balanced, we find that the degree of each node is $O(\log n)$. Since T^* is obtained from the corresponding structure for the left and right subtrees of T by the addition of one new node and $O(\log n)$ new edges, the size of T^* follows a recurrence of the form $S(m) \leq S(m') + S(m - m') + c \log m$, where c is a constant, $S(\text{constant}) = O(1)$, and m' and $m - m'$ are within a fixed ratio of each other. As is well known, any function obeying such a recurrence is of the order of $O(m)$. □

It may be interesting to note that T^* can be a nonplanar graph. Figure 3.3 is in fact an example, as it contains a K_5 among the nodes 1, 6, 7, 8, and G. (The graph can actually contain a K_m for $m = \Omega(\log n)$.) Incidentally, from now on we enforce the convention that for each node v of T, the left son v_1 will be that associated with the polygon which is locally left of $e(v)$, and similarly, the right son v_2 will be associated with the polygon locally to the right. This definition is ambiguous when $e(v)$ is horizontal, in which case we let v_1 be the son corresponding to the subpolygon below $e(v)$. Figure 3.2 was drawn following that convention. One advantage of this convention is that the two possible descendants of a node v mentioned in Lemma 3.2(ii) can be unambiguously described as the left descendant of v at level l' and the right descendant of v at level l'.

The edges (v, w) of T^* define the pairs of cutting diagonals $(e(v), e(w))$ of the polygon about which we want to know the collection of all rays cutting both $e(v)$ and $e(w)$ but avoiding the polygon P between these diagonals. If v is higher in T than w then v is an ancestor w and $e(v)$ is a bounding edge of $P(w)$. Assume (without loss of generality) that after $P(w)$ is cut by $e(w)$, the edge $e(v)$ belongs to the polygon $P(z)$ associated with the right son z of w. Both $e(v)$ and $e(w)$ are boundary edges of $P(z)$, and $P(z)$ lies in the part of P between them.

What kind of object is the collection of rays cutting two sides of a polygon while avoiding the others?

We saw in Section 2 that conditions of the form "ray $\vec{r}$ cuts segment s," or "ray $\vec{r}$ avoids segment s" both dualize to conditions requiring that the dual point D_r be in a certain double wedge. Thus, under duality, the condition that a ray cuts two sides of a polygon, but not any of the others, maps to a condition that the dual point has to be in the intersection of a set of double wedges, which is a convex polygon.

Therefore with each edge (v, w) of T^* we can associate a dual convex polygon $V(v, w)$, called a *visibility polygon*, that denotes, in dual form, the collection of all lines cutting $e(v)$ and $e(w)$ but not the portion of P between $e(v)$ and $e(w)$. Of course, $V(v, w)$ may be empty. In the notation $V(v, w)$ we always assume that v lies at a lower level than w in T, that is, v is higher up in the tree. We also follow the same convention whenever we refer to an edge (v, w) of T^*. The dual of $V(v, w)$ (clipped between $e(v)$ and $e(w)$) is called the *hourglass* at (v, w): it is bounded by two subsegments of $e(v)$ and $e(w)$, called the *exit sides*, and two concave polygonal curves joining them, whose vertices are certain reflex vertices of $P(z)$ (Fig. 3.4). A line traverses the hourglass freely, that is, does not intersect the boundary of P between the exit sides if and only if its dual point lies in $V(u, w)$. In this regard there is an equivalence between the hourglass and the visibility polygon. The equivalence is not complete, however, because of the exit sides, which are not encoded anywhere in the visibility polygon.

For convenience, let us choose as the dual transform D a degenerate polarity with the center at infinity. (This makes inclusion-testing slightly easier to discuss.) The point $p:(a, b)$ maps to the line D_p: $y = ax + b$ and the line r: $y = ax + b$ is sent to the point D_r: $(-a, b)$. (Reader, beware: this duality is not involutory.) In this way, we have two seemingly distinct cases: compare Figs. 3.4 and 3.5. (Labels refer to vertices on the left and the corresponding edges on the right.) In the 2SP these two cases are actually similar.

To determine whether a ray $\vec{r}$ traverses an hourglass freely, we can check whether the ray cuts across both exit sides and the dual of $\vec{r}$ lies in the visibility

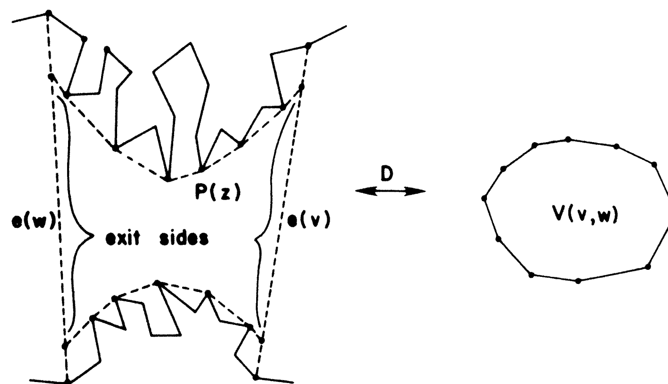

Fig. 3.4

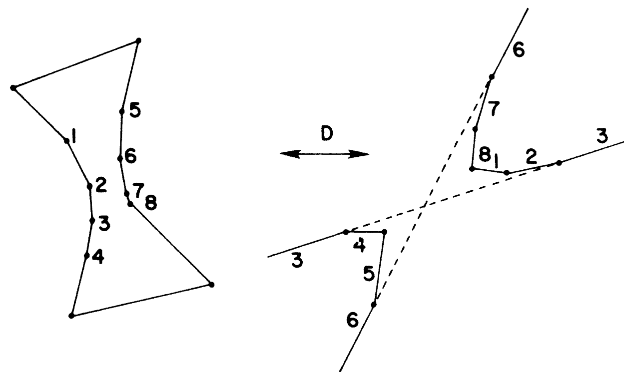

Fig. 3.5

polygon. The latter can be accomplished by means of two binary searches with respect to the x-coordinate of the point D_r. What do we mean by this? Let the term *upper* (resp. *lower*) *boundary* refer to the polygonal line running clockwise (resp. counterclockwise) from the leftmost to the rightmost vertex of the polygon (in the bounded case). Because of the monotonicity of the upper and lower boundaries we can determine their edges right above or below D_r in $O(\log n)$ time, using binary search. The unbounded case can be treated similarly. The innocent-looking fact that the binary searches always involve the same search key turns out to be of critical importance later, when the technique of fractional cascading is brought in to accelerate the searches. As a data structure, a visibility polygon is represented as a set of two arrays, each listing the vertices of the upper or lower boundaries.

Lemma 3.3. *For a node w in T consider all dual polygons $V(v, w)$ in which $e(v)$ is a diagonal bounding $P(z)$, where z is a child of w. The total size of all these visibility polygons is at most proportional to the size of $P(z)$.*

Proof. These are all distinct regions of the subdivision associated with illuminating $P(z)$ from $e(w)$, as discussed in Section 2. □

The lemma shows that all the visibility polygons associated with T^* edges whose lower end-nodes are on a particular level of T^* have total size $O(n)$, and, consequently, all the dual polygons together have total size $O(n \log n)$. (By "lower end-nodes" we mean nodes of higher level.)

3.2. *The Improved Algorithm*

We begin by describing an algorithm which uses the new structures but still takes on the order of $n \log n$ space and $\log^2 n$ query time. Then we show how to cut down each of these by a $\log n$ factor, applying incremental transformations to

the data structure. For the sake of efficiency, we assume a representation of T^* based on four types of adjacency lists. Let v be a node of T^*.

(i) The list $L(v)$ contains the left descendants of v, that is, nodes w in the left subtree of v such that the edge (v, w) is in T^*, arranged in order of increasing levels in T.
(ii) The list $R(v)$ contains the right descendants of v, that is, nodes w of the right subtree of v such that (v, w) is in T^*, arranged in order of increasing levels in T.
(iii) Let v_1 (resp. v_2) be the left (resp. right) child of v in T. The edges (u, v) of T^* fall in two categories: the set $U_l(v)$ contains the ancestors u of v such that $e(u)$ is a bounding edge of the polygon $P(v_1)$; similarly, the set $U_r(v)$ contains the ancestors u of v such that $e(u)$ is a bounding edge of $P(v_2)$. Both sets are stored as lists arranged in order of descending levels in T.

In addition, associated with each edge $(v, w) \in T^*$ we have the visibility polygon $V(v, w)$ mentioned in the previous section. We also have preprocessed for point location the triangulation of P formed by the leaves of T.

In what follows we assume that q and $\sigma^P(q, u)$ lie in different triangles of the above triangulation, otherwise ray-shooting is trivial. Let α and β be the two leaves of T associated with the triangles containing q and $\sigma^P(q, u)$, respectively. The assumption above implies that the leaves α and β are distinct, so we can turn our attention to a third node, t, defined as their nearest common ancestor. The node t has a simple geometric interpretation. It is the lowest-level node whose associated diagonal $e(t)$ is cut by the segment $s = q\sigma^P(q, u)$. To see this, we begin with the case where t is the root, for which our claim is obviously true. If t is not the root, then s lies entirely in, say, $P(v_1)$, where v_1 and v_2 are the two children of the root. But in that case, the polygon $P(v_2)$ is irrelevant, which means that the same argument can be made all over again, now substituting v_1 for the root. The claim follows by induction.

The paths in T from t to α and β also have illuminating geometric interpretations. From the standpoint of t the leaves α and β play similar roles; what we say next of β applies verbatim to α as well. Consider the path $w_0, w_1, \ldots, \beta$ in T from $w_0 = t$ to β, and assume that w_1 is not a leaf. The path itself does not tell the whole story, but it contributes nodes to a certain path of T^*, called the *β-path*, that has a compelling interpretation. From now on let $\vec{r}$ denote the ray from q in direction u. If the dual of $\vec{r}$ lies inside $V(w_0, w_1)$, the ray shoots through $e(w_1)$, and we make (w_0, w_1) the first edge of the β-path. If not, then we try $V(w_0, w_2)$, where w_2 is the unique child of w_1 in T that is connected to w_0 in T^*. (If w_1 is not a leaf, then by definition T^* must have an edge connecting w_0 to a child of w_1.) If again the dual of $\vec{r}$ does not lie in $V(w_0, w_2)$, the ray does not shoot freely from $e(t)$ to $e(w_2)$, so we must try $V(w_0, w_3)$, and so on. At some point, unless we reach β (in which case the single edge (w_0, β) constitutes the β-path), we will have a positive test, one where the dual of $\vec{r}$ is found to lie in some $V(w_0, w_h)$. Then we will make (w_0, w_h) the first edge of the β-path. We now play the same game at w_h which we did at w_0. The only ambiguity to resolve is whether the

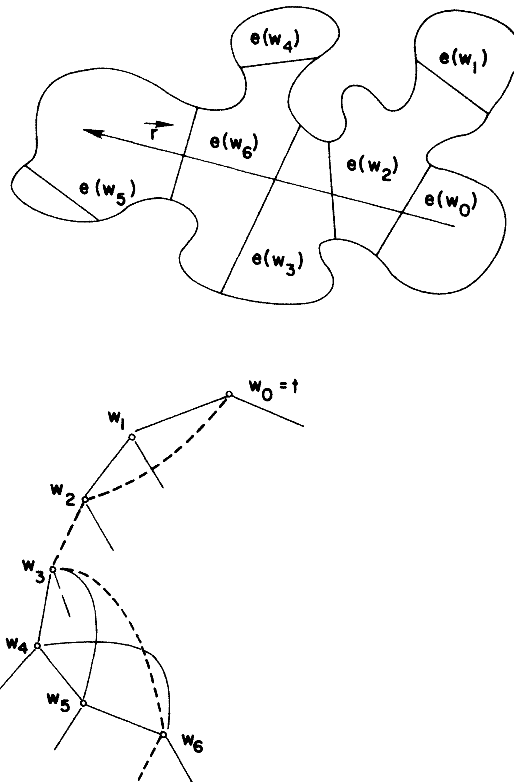

Fig. 3.6

left or right child of w_h should be considered. With our convention on the left/right orientation of the tree, it suffices to see if $\vec{r}$ shoots through $e(w_h)$ from left to right or the other way around. Continuing in this vein, we iterate through this process until we reach the leaf β. Figure 3.6 illustrates this process: the β-path consists of (w_0, w_2), (w_2, w_3), (w_3, w_6), etc.

We are now ready to describe the ray-shooting algorithm: it has three parts or *phases*. In a *start-up* phase we perform a point location to determine which triangle of the triangulation contains q, which gives us access to the leaf α. Then we go through a *root-finding* phase, which takes us from α to t, the nearest common ancestor of α and β. Finally, we enter a *descending phase* which tracks down the β-path and leads us to β. Let us now give the details of these operations.

The Ray-Shooting Algorithm

1. *Start-Up Phase.* In $O(\log n)$ time we determine the triangle $P(\alpha)$ that encloses q, using any one of the optimal point-location algorithms. We check whether the ray $\vec{r}$ exits $P(\alpha)$ via an edge of P. If so, we return the

edge in question and terminate the algorithm. Otherwise, we enter the following two phases.

2. *Root-Finding Phase.* Starting from α, we climb up the tree T to t by following a path of edges in T^*. The details are as follows. Let $e(v)$ be the boundary edge of the triangle $P(\alpha)$ through which the ray $\vec{r}$ exits the triangle. This completes the initialization of the following iterative procedure. We are in possession of a diagonal $e(v)$ that we know the ray $\vec{r}$ cuts across. Let v_1 (resp. v_2) be the left (resp. right) child of v and let j be a variable which can take on the value 1 or 2. If the ray $\vec{r}$ crosses $e(v)$ from $P(v_2)$ to $P(v_1)$ then set $j=1$, otherwise, set $j=2$. To implement this test, we check the equivalent condition (to $j=1$) that α is a descendant of v_2 in T, which can be done by checking that v_2 precedes α in preorder and α precedes v_2 in postorder. This being done, we scan the list $U_l(v)$ if $j=1$, or $U_r(v)$ if $j=2$, until a node u is found such that $\vec{r}$ freely traverses the hourglass at (u, v). We implement the test by checking whether the dual of $\vec{r}$ lies in $V(u, v)$. If there is no such node, the current node v is t and the procedure terminates. Otherwise, we reset the variable v to the node u and apply the procedure over again.
3. *Descending Phase.* We are now ready to trace down the β-path and discover the terminal triangle $P(\beta)$ and, in the process, the edge of P hit by the ray. The procedure has already been sketched out, so we just give a few complementary facts. Starting from t, we proceed toward infinity in the direction u, following the ray $\vec{r}$ from the initial position $e(t)\cap\vec{r}$. Once again we need a directional variable $j=1,2$. If the last edge (t, v) traversed in the previous phase was in the list $R(t)$, then we initialize j to 1 (meaning *left*), otherwise we set $j=2$ (meaning *right*). Set $v=t$ and begin the following iteration. If $j=1$ (resp. $j=2$) scan the list $L(v)$ (resp. $R(v)$) until a node w is found such that the dual of $\vec{r}$ lies in $V(v, w)$. If no such node w exists, then the last node reached is the leaf β and we are done. Otherwise, we update j by checking the left (resp. right) child w_1 (resp. w_2) of w: the value of j should be such that the diagonal $e(w)$ effectively separates $e(w_j)$ and $e(v)$ from each other. Checking vertex labels around the boundary of P allows us to perform this test in constant time. Our convention on left/right orientation also allows us to do the checking by geometric means. Once j has been updated, we set the variable v to w and iterate through this process.

The algorithm can be regarded as a divide-and-conquer method, and its correctness is easily shown by induction on the number of levels in the tree. For reasons we have already discussed the storage requirement is $O(n \log n)$. To answer a query involves a single point location followed by a climb up and down the tree. At most a constant number of nodes are examined per level of the tree and each examination involves checking whether a point lies inside a convex polygon of size $O(n)$, which can be done in $O(\log n)$ time. Since the tree is balanced, the query time amounts to $O(\log^2 n)$. It seems at first that we have only succeeded in trading a reasonably simple algorithm for a more complicated one with no gain in efficiency. The new data structure is perhaps not too attractive,

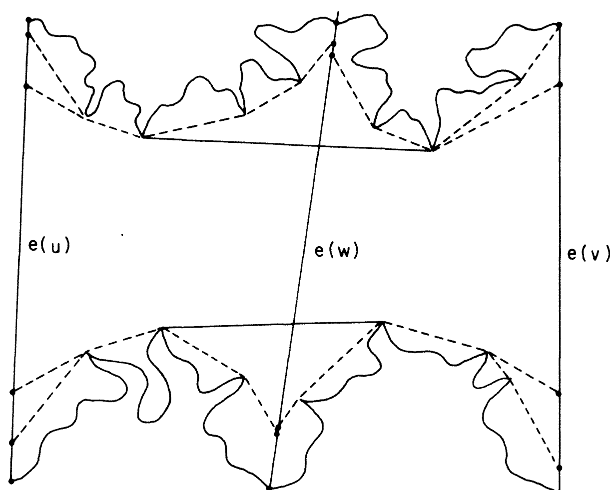

Fig. 3.7

but it is flexible. We modify it in several ways. To begin with, we must eliminate its redundancy. The visibility polygons share many edges and there is no need to represent each edge separately.

Look at Fig. 3.7. We take three nodes of T such that the three edges (u, v), (u, w), and (v, w) are in T^*. The delimiting sides of the hourglass at (u, v) are obtained from those of the hourglasses at (u, w) and (v, w) by drawing *the two common tangents* shown, and then dropping the edges replaced by the tangents. In the dual plane this means that the boundaries of $V(u, w)$ and $V(v, w)$ intersect in exactly two points (the duals of the tangents), and that $V(u, v) = V(u, w) \cap V(v, w)$. On the basis of this observation we can prove:

Lemma 3.4. *The total number of distinct edges occurring in all the visibility polygons stored in the data structure is* $O(n)$.

Proof. Assume that the lemma is true for the two subtrees of the root x of T. The result will follow if we can show that x introduces only $O(\log n)$ new edges. Indeed, the total number of distinct visibility edges will then be linear, by the same recurrence used in the proof of Lemma 3.2.

All visibilities of the form $V(x, u)$ arise for subpolygons bounded by $e(x)$ on one side and there can be at most two such subpolygons per level, one to the left and one to the right of $e(x)$. Assume now that we have already handled the visibilities $V(x, z)$ for right subpolygons, starting from the leaves and proceeding up to node w_0. Let v be the father of w_0 and assume that v is not the root. Let us ask how many new edges the visibility $V(x, v)$ can introduce.

Let w be the highest node (lowest-level) of T such that $e(w)$ separates $e(x)$ from $e(v)$ around the boundary of P. If there is no such node, then $e(x)$ and $e(v)$ are two adjacent edges of a triangle and $V(x, v)$ is of constant size. If w is defined, it is a descendant of both x and v and it is obviously unique. If $V(x, v)$

is nonempty, then neither are $V(x, w)$ and $V(v, w)$, and we have the situation depicted in Fig. 3.7 (with $u = x$). The only new vertices which $V(x, v)$ may introduce are the duals of the two common tangents of the concave chains bounding $V(x, w)$ and $V(v, w)$: these create at most four new edges. The two new cross-tangents also give rise to at most four new edges, which makes a total of eight newcomers. This accounting shows that we are introducing only a constant number of new vertices per level, so the total number of new vertices introduced by considering visibilities involving the root diagonal is $O(\log n)$. □

Lemma 3.4 says that although the visibility polygons have a total of $O(n \log n)$ edges, the number of distinct edges is only $O(n)$. To take advantage of this fact, the obvious thing to do is to store each distinct edge of the visibility polygons only once. But where? Answer: as close to the root as possible. Actually, we do not quite do that. To give a precise definition of our edge allocation strategy we need a partial order among edges of T^*: an edge (v, w) *precedes* an edge (v', w') if v is an ancestor of v' in T, or $v = v'$ and w is an ancestor of w'.

We are now ready to *make holes* in the boundary of each visibility polygon $V(v, w)$ and create a *trimmed boundary* $V^*(v, w)$. We use a coloring scheme to specify the holes. Initially, every edge of T^* has its own *distinct* color which is also the color of the boundary of its visibility polygon. At the end, every visibility polygon will have its boundary colored in various ways. The monochromatic polygonal lines in which the boundary is partitioned are called *strands*. A strand of $V(v, w)$ whose color remains what it was initially is called *resident*: it is stored entirely in the trimmed boundary $V^*(v, w)$. A nonresident strand of $V(v, w)$ is stored in the trimmed boundary $V^*(x, y)$ associated with the unique T^* edge (x, y) of the same color. Let $v_1, v_2, \ldots, v_k$ be the vertices of a nonresident strand from left to right (strands will always be monotone in the x-direction[2]). To preserve consistency we store the edge (v_1, v_k) in $V^*(v, w)$, but since it may not be on the boundary of any visibility polygon we call it a *ghost*. Before we elaborate on the implementation of the data structure it is best to describe the coloring process of $V(v, w)$.

Initially, the entire boundary is monochromatic. Let e be an edge of the upper boundary of $V(v, w)$. We define the *signature* of e to be the vertex of P whose dual is the line of support of the segment e. Let p be an arbitrary point of e (distinct from an endpoint). Because of the tree structure of T the set of T^* edges which precede (v, w) is totally ordered. Among those edges let E be the subset of edges whose visibility polygons have a bounding edge f on their upper boundaries such that (i) p lies on the edge f and (ii) e and f have the same signature. Because of the total order in E, we can identify a first edge (x, y). Our strategy is to color p after (x, y). Needless to say, lower boundaries are colored just the same way.

The coloring scheme is almost saying "*color every visibility edge after its highest occurrence in the tree*," although not exactly, because a given edge may be broken

[2] We are gliding over the fact that visibility polygons may wrap around infinity and come back; this may disturb our argument a little but not enough to warrant a separate discussion.

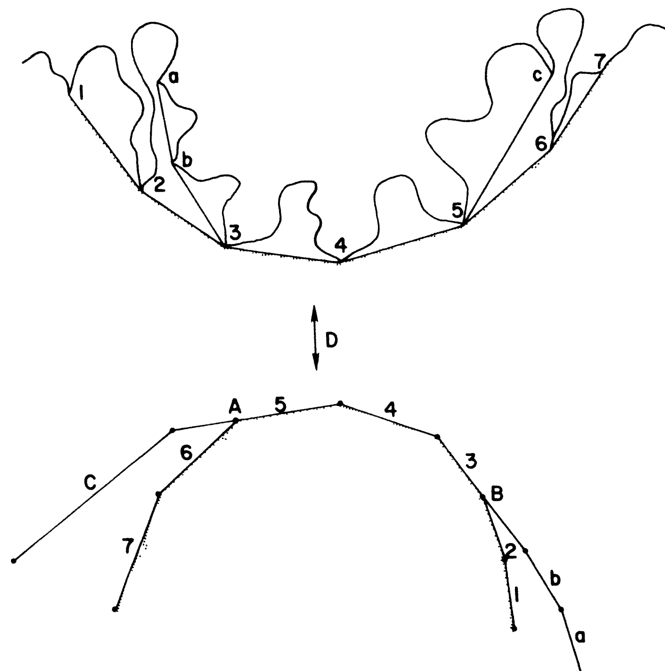

Fig. 3.8

up into several parts. It is easy to see that two visibility polygons may have collinear edges without common edges (Fig. 3.8). This means that strands do not have to be edge sequences of the visibility polygons: they may run between points which are not vertices of the visibility polygons. For example, in Fig. 3.8 the upper boundary with edges labeled $a, b, 3, 4, 5, c$ may have a strand running from point A to point B, although these points are not vertices of the visibility polygon in question. Lemma 3.4 tells us that we could opt for the simpler strategy of storing every edge as high in the tree as we can and still ensure linear space. Why do we not do that? The reason is that we cannot afford duplication of points (the reader may not be able to appreciate this fine point now, so be patient). With our scheme, if we were to pick a vertex u of P and consider all the trimmed boundary edges with u for signature, we would find that, of course, all these edges are collinear, but most important, no two of them overlap (outside of their endpoints). This property is crucial to ensure logarithmic query time.

The data structure used to represent a trimmed boundary consists of two arrays, one for the upper boundary and the other for the lower one. The arrays are sequences of records: *regular* records store edges of resident strands, and *ghost* records store the edges into which nonresident strands are reduced along with pointers to the T^* edges where they are effectively stored (Fig. 3.9). Actually, ghosts simply point to the first element of the trimmed boundary to which they correspond. The size of the resulting data structure is proportional to the total

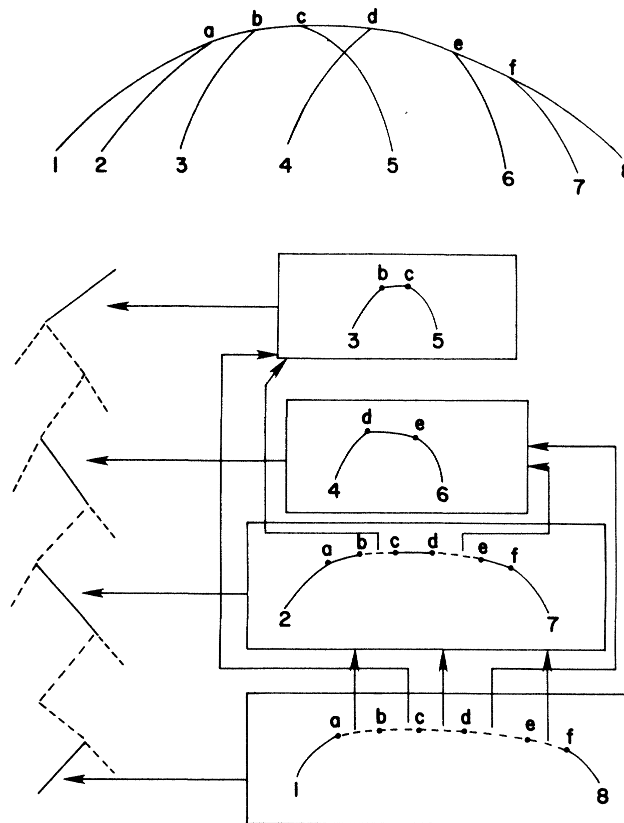

Fig. 3.9

number of distinct visibility edges plus the number of ghost records plus n. We know from Lemma 3.4 that the number of distinct edges is $O(n)$. How about the number of ghosts? It is easy to prove that a given trimmed boundary has $O(\log^2 n)$ ghosts. As it turns out, this upper bound would be sufficient for our purposes later, although it would require a little extra work from us. But we can do better, as our next result shows.

Lemma 3.5. *The number of ghost records in any given trimmed boundary is* $O(\log n)$.

Proof. Without loss of generality let us restrict our attention to the upper boundary of the visibility polygon $V(v, w)$. Let C be the corresponding (upper) boundary of the hourglass at (v, w). Imagine rolling an infinite line L around C clockwise to that L traverses the hourglass freely at all times. The rotational span of L is less than π, so let $L(\theta)$ denote the line L for a given angular slope θ; we choose a reference system so that $\theta \in [0, \pi)$. Let H be the set of hourglasses at

edges of T^* that precede (v, w). For a given θ, let $\nu(\theta)$ be the set of hourglasses of H which are traversed freely by $L(\theta)$ and have $C \cap L(\theta)$ on their upper boundaries. The number of ghosts in the trimmed upper boundary at (v, w) cannot exceed 1 plus the number of changes in the function $\nu(\theta)$, as θ varies from 0 to π. Let (x, y) be an edge of T^* that contributes an entry to $\nu(\theta)$. A simple case-analysis shows that if x is an ancestor of v in T, then y is an ancestor of v or v itself. This implies that the hourglasses in $\bigcup_{0 \le \theta < \pi} \nu(\theta)$ have a total of $O(\log n)$ distinct exit diagonals (there may be more exit sides since one diagonal can contribute many of them). We call these diagonals candidates. Of course, this still leaves open the possibility of $\Theta(\log^2 n)$ hourglasses in $\bigcup_{0 \le \theta < \pi} \nu(\theta)$. To see that this cannot be the case, observe that $L(\theta)$ can intersect a given candidate diagonal, and be a line of visibility between $C \cap L(\theta)$ and that diagonal, over at most two angular intervals. This implies that the number of value changes in the function ν is at most proportional to the number of candidate diagonals, which proves the lemma. □

Since T^* has $O(n)$ edges, an immediate corollary of the lemma is that the data structure requires $O(n \log n)$ storage. What progress have we made? Although the bound is still unsatisfactory there is cause for optimism. Indeed, the only snag in the way of a linear bound is the possibility of having too many ghosts. But Lemma 3.5 shows that a given T^* edge cannot have more than a logarithmic number of them. Pruning the bottom levels of the tree T will take care of this problem, as we shall see later. (This can be avoided, as it turns out, but pruning is in general a good, practical idea, anyhow.) Our new data structure still supports ray-shooting queries. The algorithm is only slightly more complicated. The one significant difference is that whenever a search lands in a ghost record, we must carry it on in the trimmed boundary to which it points. This time around, by definition, the search will succeed. The query time is trivially $O(\log^2 n)$.

The first item on the agenda is to cut down the query time to $O(\log n)$. To speed up the query-answering process, we rely heavily on a technique for iterative searching called *fractional cascading* [4]. We assume that the reader is familiar with this technique. The bulk of the ray-shooting algorithm involves iterated dictionary look-ups in $O(\log n)$ *catalogs*. In the fractional-cascading terminology a catalog is the name given to a sorted linear list. In this case, the catalogs are the trimmed upper and lower boundaries associated with the edges of T^*. To simplify our discussion, we deal only with trimmed upper boundaries and treat $V^*(v, w)$ as though it consisted only of the trimmed upper boundary of $V(v, w)$. Of course, it is understood that whatever we say also applies to the trimmed lower boundaries. Because our data structure consists of a collection of catalogs associated with the edges of a graph it is possible to apply fractional cascading to it. Technically speaking the association should involve the nodes of the graph and not its edges. To overcome this difficulty we could modify T^* by adding a dummy node in the middle of each edge (v, w) and associating the trimmed upper boundary $V^*(v, w)$ with it. For consistency the original nodes of T^* are assigned empty catalogs.

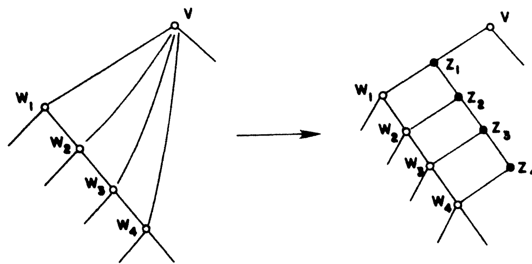

Fig. 3.10

There is still one problem: fractional cascading is most efficient when the underlying graph has bounded degree, which is not the case with T^*. Let us return T^* to the state in which it was prior to the addition of the dummy nodes. For each node $v \in T$, consider the sets $L(v)$ and $R(v)$ in turn, and add respectively $|L(v)|$ and $|R(v)|$ nodes. We describe the procedure on $L(v) = (w_1, w_2, \ldots, w_k)$. It involves replacing the edges $(v, w_1), \ldots, (v, w_k)$ by (v, z_1), (z_i, w_i), and (z_i, z_{i+1}), for each i $(1 \le i \le k-1)$, and (z_k, w_k). See Fig. 3.10. Each node z_i is brand-new: it is called the *catalog node* of the T^* edge (v, w_i) and is assigned as catalog the trimmed upper boundary $V^*(v, w_i)$. If $V(v, w_i)$ is empty, so is the catalog assigned to z_i. All the other nodes are assigned empty catalogs. This takes care of the descendant neighbors of v in $L(v)$ and $R(v)$, but we are not done yet.

What about $U_l(v)$ and $U_r(v)$? We follow the same principle. Let $w_1, \ldots, w_k$ be the nodes of $U_l(v) \cup U_r(v)$ in decreasing order level-wise (i.e., in tree-ascending order), and let z_i be the catalog node of the pair (w_i, v). For each i $(2 \le i \le k)$, we remove the edge (z_i, v) and replace it by (z_i, z_{i-1}). As usual, we denote edges by pairs indicating their natural top-down orientation. Thus, we refer to the edge (z_i, z_{i-1}) and not (z_{i-1}, z_i) because w_i is an ancestor of w_{i-1}. Figure 3.11 shows the transformation of T^* along a path of five nodes with the complete graph on it. Black dots represent catalog nodes with labels indicating their bijection with the edges of T^*. This transformation ensures that the resulting graph, denoted G, has maximum degree four.

How do we navigate in T^*? To carry out our ray-shooting routine we need to be able to scan $L(v)$ or $R(v)$ as well as $U_l(v)$ and $U_r(v)$ and interrogate the

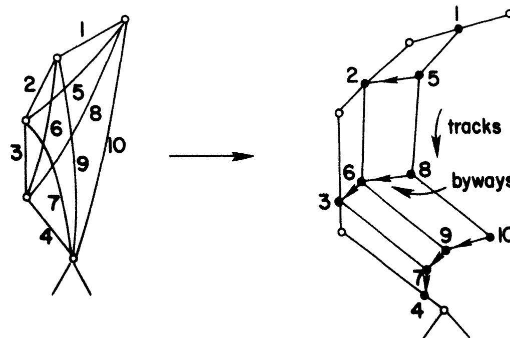

Fig. 3.11

trimmed boundaries stored at the catalog nodes of the corresponding edges. The nodes of T^* are nodes of G but the converse is not true. However, to every edge (v, w) of T^* corresponds at least one path from v to w in G. Let $L(v) = (w_1, \ldots, w_k)$ and let z_i be the catalog node of the T^* edge (v, w_i). The path $z_1, z_2, \ldots, z_k$ in G can be thought of as a *track*. Tracks can be crossed by using *byways*—see paths marked with arrows in Fig. 3.11. To traverse an edge (v, w) of T^* (and interrogate its visibility structure) we go from v to w in G by walking down a track first and then connecting to w via the appropriate byway. This allows us to carry out the descending phase in G with no difficulty whatsoever. The time taken to visit a given edge (v, w) of T^* is proportional to the level differential between v and w. To scan down $L(v)$ is not as costly, however. We easily verify that to visit $z_1, \ldots, z_l$ $(l \leq k)$ in this order takes $O(l)$ steps. Therefore we can carry out in G the entire descending phase of the ray-shooting algorithm while incurring the cost of only $O(\log n)$ catalog look-ups. Furthermore, the order of the nodes visited follows what is called a *generalized path* in the fractional-cascading terminology. This means that each node visited, except the first one, is adjacent to a node visited earlier.

Now how about $U_l(v)$ and $U_r(v)$? To traverse the byway from v upward is the equivalent in G of scanning the union of $U_l(v)$ and $U_r(v)$. Appropriate labeling of the nodes allows us to distinguish between $U_l(v)$ and $U_r(v)$. Again, the complexity of scanning $U_l(v) \cup U_r(v)$ is essentially proportional to the number of T^* edges examined. In other words, what we just said of the descending phase also applies to the root-finding phase. One final point: the pointers stored in the ghost records of the catalogs should point not to the edges of T^* but to their associated catalog nodes in G. As it turns out, it is best to keep both T^* and G around. Admittedly, there is no need for T^* right now, but its time will soon come.

What conclusions are we to draw at this point? The entire ray-shooting algorithm can be ported to the graph G augmented with its various catalogs. Assume the existence of an oracle that provides a constant-time answer to any search which happens to land in a ghost record. Recall that when a search terminates in a ghost it must be pursued in the boundary list pointed to by the record in question. We are not quite ready to do this extra work in constant amortized time, so let us punt and assume that someone else will do the job for us at no cost. Then, nice things begin to happen. The work involved in answering a ray-shooting query fits squarely into the iterative search framework of fractional cascading and the speed-up technique can be applied. This cuts down the query time to $O(\log n)$.

3.3. *Giving Life to the Oracle*

It remains for us to dream up a method which answers oracle queries in constant amortized time and does not add too much storage. One problem is that the T^* edges referenced by the ghost records of a given trimmed boundary can be scaringly random-looking (see the Appendix). To be sure, the edges hit by the oracle have their vertices on the path from the root of T to the leaves α or β.

There can be $\Theta(\log^2 n)$ such edges, however, and if the oracle were to be asked to deal with an arbitrary subset of $\Theta(\log n)$ of them we would be in serious trouble. As a matter of fact the best that fractional cascading could then offer would be $O(\log n \log\log n)$ query time, where the factor $\log\log n$ comes from the logarithm of the degree of the underlying graph.

Fortunately, the power of fractional cascading has not been exhausted yet. Define the range of a catalog to be the interval formed by its smallest and largest key. The bounded degree condition can be relaxed into the following condition: given any node v and any real x in its range, the number of nodes adjacent to v in the catalog graph, whose ranges contain x, must be bounded above by a fixed constant. This condition is similar to the *locally bounded degree condition* of [4], though slightly stronger. As is shown in [4], the results of fractional cascading basically remain unchanged under this weaker assumption.

How does this relate to our problem? We know that the only edges interrogated by the oracle connect two nodes on the path π_α from the root to α, or on the path π_β from the root to β. Our plan is to interrogate every relevant edge along these paths before the oracle has a chance to do so. By "relevant" we mean edges whose visibility polygons have nontrivial information to offer. We will show that there are only $O(\log n)$ relevant edges (as opposed to a total of $\Theta(\log^2 n)$). Before going any further, let us define this notion of relevance more precisely.

Recall that we are dealing only with trimmed upper boundaries. An edge (v, w) of T^* is *relevant* if both v and w are nodes of $\pi_\alpha \cup \pi_\beta$ and the trimmed upper boundary $V^*(v, w)$ contains a regular edge which lies right above the dual of the ray $\vec{r}$. This edge is called the *clue of* (v, w) (when $\vec{r}$ is understood). To lie right above a point means that the point lies in the vertical strip defined by the endpoints of the edge and below the line supporting the edge. Every regular visibility edge ever needed during ray-shooting is relevant, but the converse need not be true.

In the case of π_α, we know the path ahead of time, so we can compute the relevant edges and all their clues prior to the root-finding phase. In the case of π_β we have to proceed incrementally. Every time we go from a node of π_β to one of its children v we compute all the relevant edges of T^* of the form (w, v) as well as their clues. As follows from an observation made in the proof of Lemma 3.5, clues will thus be available before they are needed. Whenever we compute a clue, we stash it away along with the catalog node of G corresponding to the edge in question. In this way, the oracle will get its desired answer by a simple look-up.

Let us summarize what we are trying to do. For simplicity we can assume that we have two distinct data structures, (i) the augmented graph G and (ii) the graph T^* augmented with special features—yet to come—to support the computation of relevant edges and their clues. Prior to the root-finding phase, the clues of all the relevant edges of π_α are computed and stored in the corresponding catalog nodes of G. In this way, the root-finding phase can proceed in G as described earlier. Every call on the oracle can be replaced by a simple reading of the clue in the catalog node pointed to by the current ghost record. The descending phase is very similar. The only difference is that all the clues cannot

be computed ahead of time. Every time we descend to a new node v, all the clues of the relevant edges (w, v) are computed and stashed away in G, right alongside the edges in question. As a matter for fact, for consistency, we can do the same thing for π_α as well. Then, in all cases, we are repeatedly confronted with the following situation. Let v be our current node and let $(w_1, w_2, \ldots, w_k)$ be the nodes of $U_l(v) \cup U_r(v)$: identify the relevant edges among $(w_1, v), \ldots, (w_k, v)$, and compute and store their clues.

To make this problem conform with the iterative framework of fractional cascading, once again we transform T^* by removing all the edges (w, v) which are not parent/child relationships and we attach to each node v a path of k nodes, $z_1, \ldots, z_k$. Each z_i is associated with the trimmed upper boundary $V^*(w_i, v)$ which forms its catalog. This gives us a valid catalog graph all right, but one quite undesirable in several ways. For one thing, we may not be able to afford the traversal of $z_1, \ldots, z_{k-1}$ if, say, only z_k is relevant (an abuse of notation to say that (w_k, v) is relevant). To avoid this problem, we take one bold step and break down the catalog of each z_i, that is, the trimmed upper boundary $V^*(w_i, v)$. To do so, we remove all the ghosts from the list and make a separate catalog of each connected piece. Each new catalog is associated with a separate node $z_i^{(j)}$. In this way we trade z_i and its catalog for a collection of nodes $z_i^{(1)}, z_i^{(2)}, \ldots$, whose catalogs partition the catalog of z_i. Finally, we put an edge between the nodes $z_i^{(a)}$ and $z_j^{(b)}$ $(i<j)$ if there exists a real x in the range of both nodes but not in the range of any $z_l^{(c)}$ such that $i<l<j$. We include $z_0^{(1)} = v$ in this specification and assign the empty catalog to it with range equal to $\mathfrak{R}$. Admittedly, this transformation may sound a little mysterious. A picture will help (Fig. 3.12). Any node may have large degree but its *local* degree around a certain key does not exceed some constant. Note that there are no edges connecting the nodes associated with the catalogs coming from the same boundary list.

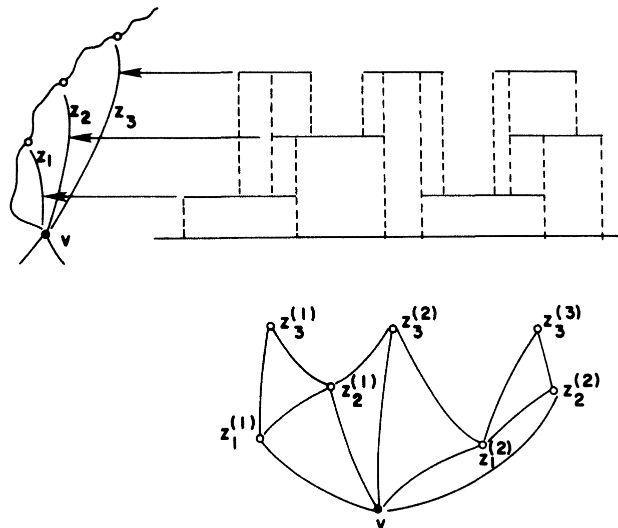

Fig. 3.12

It is immediate to check that the graph satisfies the weaker degree condition stated earlier. We can therefore apply fractional cascading to it. The reader will recognize in this construction the so-called *hive-graph* of Chazelle [2]. When answering a ray-shooting query, the nodes z_i that are relevant correspond to the z_i^j's whose ranges include the x-coordinate of the dual of $\vec{r}$ (that is, the slope of $\vec{r}$). Referring to Fig. 3.12, these are obtained by intersecting the horizontal segments with the vertical line $x = a$, where a is the x-coordinate of the dual of $\vec{r}$. Fractional cascading will let us do that in a straightforward manner in time proportional to the total number of relevant nodes z_i plus $\log n$. Putting our results together, we have a ray-shooting algorithm that requires $O(\rho + \log n)$ time, where ρ is the number of relevant edges in T^*. How large can ρ be?

Lemma 3.6. *The number of relevant edges with respect to a given ray-shooting query is* $O(\log n)$.

Proof. Recall that an edge (v, w) of T^* is relevant if both v and w are nodes of π_α or π_β and the trimmed upper boundary $V^*(v, w)$ contains an edge that lies right above the dual of $\vec{r}$. This puts an obvious upper bound of $O(\log^2 n)$. on ρ. Actually, if we substituted $V(v, w)$ for $V^*(v, w)$ in the definition of relevance, this upper bound would be tight. This fact, which we leave as an exercise, shows that trimmed boundaries are good not only for the space they save but also for the ray-shooting time. To prove the lemma, we may restrict ourselves to the relevant edges along only one of the two paths π_α or π_β. Call this path π and let p be the dual point of $\vec{r}$. We assume that $\vec{r}$ is not parallel to the line passing between any pair of vertices of P. This assumption—which can be easily relaxed—implies that p is not a vertex of any visibility polygon. Recall that we took great pains to ensure that trimmed boundary edges with the same signature may not strictly overlap. This ensures that no two relevant edges joining nodes of π can have clues with the same signature.

We are now ready to collect the rewards of our efforts. Let us say that a vertex t of P is *exposed* if there exists a line segment parallel to $\vec{r}$ which contains t and traverses freely the hourglass of a relevant edge of T^* from one exit side to the other (Fig. 3.13). It follows that ρ cannot exceed the number of exposed vertices. But it is easy to see that there are only $O(\log n)$ exposed vertices. Why? Cut the boundary of P at the endpoints of the cutting edges associated with the nodes of π. This produces $O(\log n)$ connected polygonal pieces. Now assume that one of these pieces contains two exposed vertices t and t'. Let AB and $A'B'$ be the two segments that witness the "*exposedness*" of t and t', respectively. From elementary geometric considerations it is clear that the part of the polygonal piece running between t and t' must contain one vertex among A, B, A', B' (Fig. 3.14). Consequently, the polygonal piece in question is not connected, which produces a contradiction and proves our claim. □

All the pieces are now together. At long last, we have obtained an $O(n \log n)$-size data structure for ray-shooting in $O(\log n)$ time. To reduce the space

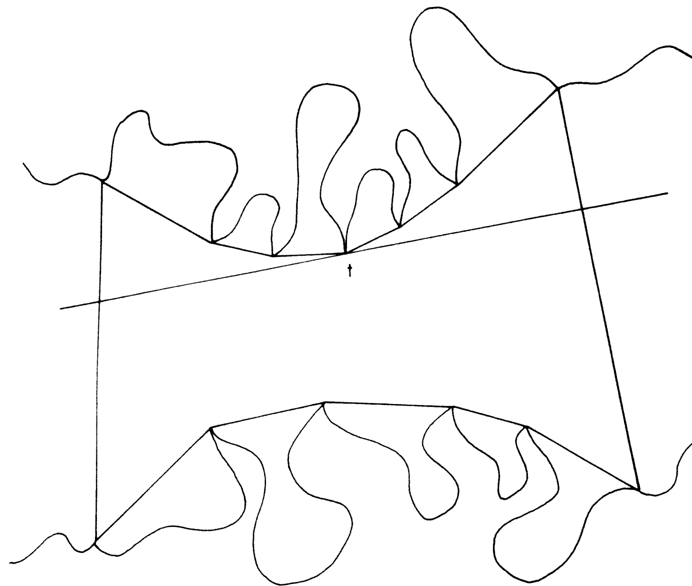

Fig. 3.13

requirement to $O(n)$ is not difficult. We can use a standard tree-pruning trick [13], [9]. Remove from T all the nodes with fewer than $\log n$ descendants and remove all the edges of T^* incident to these nodes. Since T is built from a balanced decomposition of P, this leaves us with a graph of $O(n/\log n)$ nodes. Following Lemma 3.2, this graph has at most $O(n/\log n)$ edges. To fit this new scenario entails modifying the data structure in a straightforward fashion. As we noticed previously, the only reason why we did not achieve linear storage earlier was the presence of ghosts. But, from Lemma 3.5, ghosts cost only $O(\log n)$ storage per edge of the graph, therefore the reduced data structure requires $O(n)$ storage. Of course, the only trouble is that instead of triangles, we must now deal

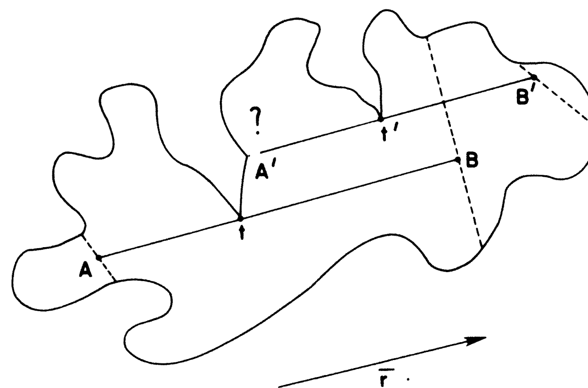

Fig. 3.14

with polygons of $O(\log n)$ edges. But these are one-shot costs during ray-shooting (pun intended!). The overhead is $O(\log n)$ and does not add to the asymptotic complexity of the algorithm.

The last point to address is the construction of the data structure. The simplest strategy is to triangulate the polygon and preprocess it for efficient planar point location. Then we build the (reduced version of the) graph T^* and all the visibility polygons in full. If we proceed bottom-up we can use the fact that any visibility polygon is the intersection of two previously computed polygons A and B, and thus can be computed in time linear in the added size of A and B. This ensures that all the preprocessing so far can be done in $O(n \log n)$ time. Continuing in the same vein we can trim the visibility polygons by computing their signatures in a first stage. Then for each signature we sort the endpoints of the corresponding visibility edges (recall that all these edges are collinear). This allows us to break down each edge into intervals and *color* each interval appropriately. Replacing colors by pointers we obtain G and its catalogs in $O(n \log n)$ time. Fractional cascading takes linear time. Finally, each hive-graph can be computed in a sweep-line fashion in $O(n \log n)$ time.

Theorem 3.7. *There exists an $O(n)$-space data structure representing a simple polygon P which can be computed in time $O(n \log n)$ and which, given a pair (q, u) of a point q and a direction u, can be used to find the first edge of P hit by the ray from q in the direction u in time $O(\log n)$.*

4. Extensions to Intersection Problems

The previous result can be extended in two ways. First, having found the first intersection of P with the ray (q, u), we can proceed to find the next one in the same manner, by just decomposing the exterior of P into a number of simple polygons and applying the above preprocessing to each of these polygons. Thus in fact we have a technique for computing all k intersections between a line segment and a simple polygon in time $O((k+1) \log n)$. This algorithm is unlikely to be the method of choice, however. Indeed, there exists a much simpler linear-size data structure for this problem with query time $O((k+1) \log(n/(k+1)))$ [5].

Finally, we observe that our ray-shooting algorithm generalizes easily from simple polygons to arbitrary subdivisions of the plane into simply connected polygonal regions. How so? Break each cycle in the planar graph by duplicating vertices (Fig. 4.1). Perform this duplication process until the set of edges forms a single tree. Then perform a depth-first search traversal of the tree. Each edge is visited twice, so we will duplicate it on the second visit. Here duplication means the creation of an identical edge distinct from but very close to the original one. Figure 4.1 illustrates this two-stage process. The set of edges can be now made the boundary of a simple polygon (very thin, perhaps, but still...). Another approach is to build separate shooting structures for each connected region of the subdivision, and then use point location to identify the region to which the ray-shooting query should be applied.

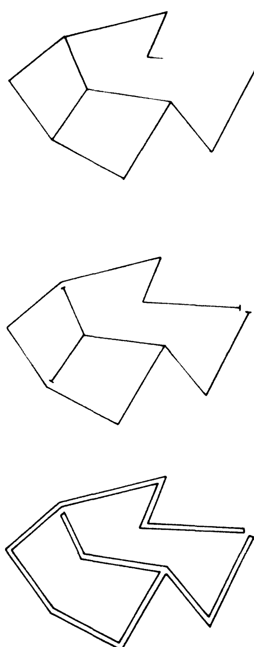

Fig. 4.1

Acknowledgments

We wish to thank the anonymous referees for their useful comments and suggestions.

Appendix

We wish to show that the ghosts which appear in ray-shooting may point to almost arbitrary T^* edges. Consider the polygon depicted in Fig. A.1 and its associated tree T. The vertical segments represent the cutting edges along a path from the root to some node down the tree. The nodes of the path are labeled 1-12 in level-ascending order. Think of 1 as, say, the root diagonal and 12 as a node near the bottom of the tree. We have also drawn the path 1-12 with the edges connecting nodes of the path. Consider now a horizontal ray-shooting query. It may happen that the trimmed boundaries $V^*(7, 8)$, $V^*(9, 10)$, $V^*(11, 12)$ are interrogated during searches and that the desired answers (the clues) are the edges e_a, e_b, e_c whose signatures are the vertices a, b, c, respectively. As it turns out, all three edges are ghosts: e_a, e_b, e_c are actually stored in $V^*(1, 6)$, $V^*(2, 5)$, $V^*(3, 4)$, respectively. Notice that we have the complete graph on the nodes 1-7. The ghost edges have to be retrieved among some of the T^* edges of this graph. To see how arbitrary this subset can be, observe that instead of the path $1, 2, \ldots, 12$ we could have a path $\pi, 7, 8, 9, 10, 11, 12$, where π is *any*

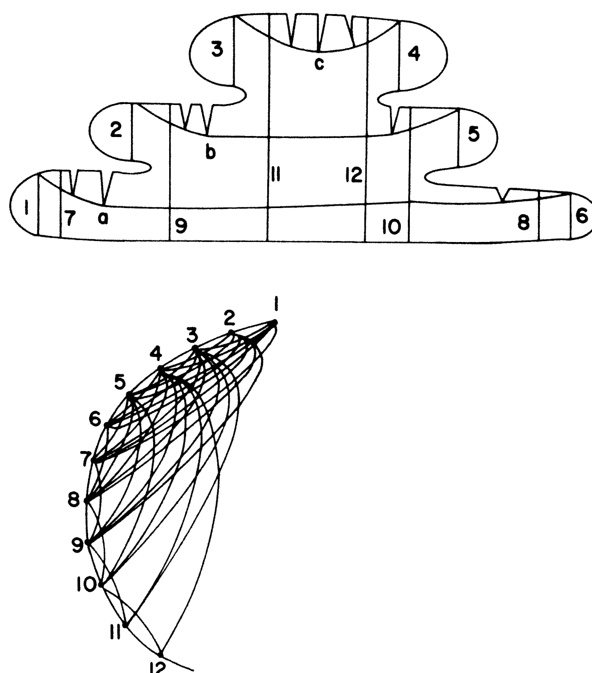

Fig. A.1

permutation of $(1, 2, 3, 4, 5, 6)$. The point is that by adding or deleting vertices to P we can basically force any order we want among the cuts $1, \ldots, 6$. For example, we can always make 6 the root of T by adding sufficiently many dummy vertices along the boundary enclosing the label 6 in Fig. A.1.

References

1. Chazelle, B. A theorem on polygon cutting with applications, *Proc. 23rd Ann. IEEE Symp. Found. Comput. Sci.* (1982), 339–349.
2. Chazelle, B. Filtering search: a new approach to query-answering, *SIAM J. Comput.* **15** (1986), 703–724.
3. Chazelle, B. Computing on a free tree via complexity-preserving mappings, *Algorithmica* **2** (1987), 337–361.
4. Chazelle, B., Guibas, L. J. Fractional cascading: I. A data structuring technique, *Algorithmica* **1** (1986), 133–162.
5. Chazelle, B., Guibas, L. J. Fractional cascading: II. Applications, *Algorithmica* **1** (1986), 163–191.
6. Edelsbrunner, H., Guibas, L. J., Stolfi, J. Optimal point location in a monotone subdivision, *SIAM J. Comput.* **15** (1986), 317–340.
7. El Gindy, H. A. An efficient algorithm for computing the weak visibility polygon from an edge in simply polygons, unpublished manuscript, McGill University, 1984.
8. Guibas, L. J., Hershberger, J., Leven, D., Sharir, M., Tarjan, R. E. Linear-time algorithms for visibility and shortest-path problems inside triangulated simple polygons, *Algorithmica* **2** (1987), 209–233.
9. Guibas, L. J., Hershberger, J. Optimal shortest-path queries in a simple polygon, *Proc. 3rd Ann. ACM Symp. Comput Geom.* (1987), 50–63.

10. Guibas, L. J., Ramshaw, L., Stolfi, J. A kinetic framework for computational geometry, *Proc. 24th Ann. IEEE Symp. Found. Comput. Sci.* (1983), 100–111.
11. Lee, D. T., Lin, A. Computing the visibility polygon from an edge, unpublished manuscript, Northwestern University, 1984.
12. Tarjan, R. E., Van Wyk, C. An $O(n \log\log n)$-time algorithm for triangulating simple polygons, *SIAM J. Comput.*, **17** (1988), 143–178.
13. van Emde Boas, P., Kaas, R., Zijlstra, E. Design and implementation of an efficient priority queue, *Math. Systems Theory* **10** (1977), 99-127.

Received June 14, 1986, and in revised form June 18, 1988, and January 18, 1989.