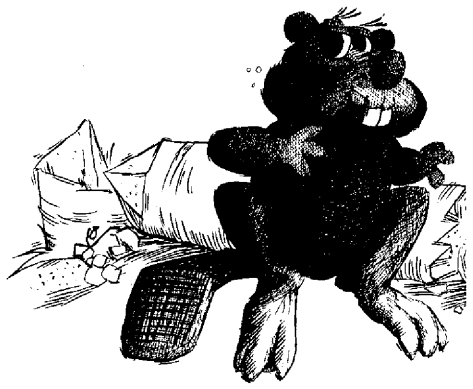

The Contour Problem for Polygons

Thomas Ottmann
Derick Wood

Data Structuring Group
CS-84-33

October, 1984

THE CONTOUR PROBLEM FOR POLYGONS[1]

Thomas Ottmann[2]

Derick Wood[3]

ABSTRACT

The union of a set S of p, not necessarily disjoint, simple polygons in the plane determines a set of disjoint polygons, possibly having holes. We present a plane-sweep algorithm to compute the edges of the resulting disjoint polygons, that is their *contour*, which runs in $O((n+k)\log n)$ time and $O(n)$ space, where n is the total number of vertices in the polygons and k is the number of edge intersections among the polygons. The space requirement is reduced to $O(p)$ if the polygons are convex.

Although the result is new, the main focus of the paper is on a general translation principle which yields the current solution. Given a plane-sweep algorithm for some problem for the, highly restricted, isothetic, that is, rectilinear case, this algorithm can be translated into one for the same problem for the non-isothetic case. The translation principle enables a two-step solution to non-isothetic problems to be obtained; first, the simpler isothetic problem is treated and, second, the translation principle is invoked to yield the final solution.

1. INTRODUCTION

Many of the problems studied in computational geometry have the following format:

Given a set S *of objects of type* T *compute the function* F(S).

---

1 The work of the first author was supported by DFG Grant Ot 64/4-2, and that of the second by a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692. The work was carried out while both authors were visiting the Computer Science Department of the University of Helsinki.

2 Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, D-7500 Karlsruhe, W. Germany.

3 Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

For example T consists of simple polygons in the plane and $F(S)$ is the set of intersecting pairs in S, or T consists of simple isothetic or rectilinearly-oriented polygons and $F(S)$ is the set of connected components in S. Typically solution strategies for problems of simple polygons and simple isothetic polygons are quite different, even when both are based on the plane-sweep paradigm, since in the latter use is made of the restricted nature of isothetic polygons. In [W, OW1, OW2] a method was introduced for bridging the gap between solution strategies for two problems where the only difference is that T consists of isothetic polygons in one case and non-isothetic polygons in the other. In overly simplistic terms [W, OW1, OW2] present a method of translating plane-sweep solutions of isothetic problems into plane-sweep solutions of the corresponding non-isothetic problems. The translation principle is based on two related ideas: *zig-zags* and *semi-dynamic search structures.* The advantage of the translation principle is that solutions obtained for isothetic problems, which are much easier to obtain, can be translated into solutions for non-isothetic problems with only standardized modifications. This step-wise approach to solving non-isothetic problems is clearly advantageous over the creation of a solution from scratch. Its only disadvantage is that the time complexity of such a solution is bounded from below by $\Omega((n+k)\log n)$, where n is the total number of vertices in the polygons and k is the number of edge intersections among the polygons. This is simply because it depends on the algorithm of [BO] and [Br] for finding the intersections. Until recently this was more than acceptable since $O((n+k)\log n)$ time was the best upper bound known for finding the intersections in the first place. Indeed in [NP] a number of problems are solved with this upper bound. However the recent paper of [C] has demonstrated an $O(n\log^2 n+k)$ time upper bound for the edge intersection problem and it leads us to suspect many non-isothetic problems will yield solutions with better worst-case bounds based on the approach in [C].

Notwithstanding the results in [C] we feel that the translation principle is important *in its own right* as a contribution to programming methodology. Moreover we anticipate that a similar translation principle can be obtained based on the results in [C]. We examine a particular problem, namely T consists of simple polygons and $F(S)$ is the set of disjoint polygons defined by the union of the polygons in S, the *contour problem.* As far as we are aware this problem has not been treated explicitly in the literature, although a solution could perhaps be obtained from the results in [NP]. In the isothetic case it has received much attention after its introduction in [LP], see [G1, G2, G3, Wo]. We translate the solution given in [Wo] for the isothetic case, which is time- and space-optimal, to give a solution for the non-isothetic case which requires $O((n+k)\log n)$ time and $O(n)$ space, where n is the total number of vertices in the given polygons and k is the number of edge intersections.

The paper consists of two further sections. In Section 2 the use of zig-zags is motivated by way of semi-dynamic structures. The contour algorithm is presented in Section 3 paralleling its presentation in [Wo] for isothetic polygons.

2. SEMI-DYNAMIC STRUCTURES AND ZIG-ZAGS

When computing, off-line, properties of sets of isothetic or rectilinear polygons in the plane the plane-sweep paradigm has become a valued tool for deriving efficient and often optimal algorithms. Such an approach often makes use of *semi-dynamic search* data structures, that is data structures whose *skeletal*, or underlying, structure is computed before the plane-sweep begins. Insertions into and deletions from a semi-dynamic search structure do not change its skeletal form, updates only modify information at its nodes. For example the segment tree of [BW], the tile tree of [E] or [Mc], the layered segment tree of [VW], and the visibility tree of [Wo]. Semi-dynamic structures can be used when treating such off-line problems because they satisfy the following conditions:

(1) The values to be inserted and subsequently deleted are known in advance.

(2) The values, which correspond to endpoints of vertical line segments and to horizontal line segments, are totally ordered with respect to the initial position of the sweep line, see Figure 2.1, and this order never changes.

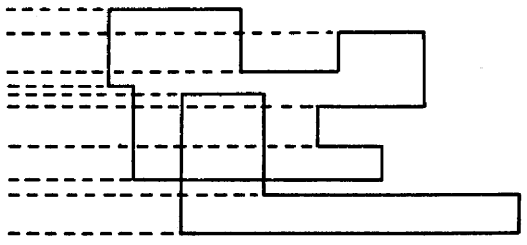

Figure 2.1

However when we consider sets of (non-isothetic) polygons in the plane Condition (2) does not hold as is easy to see in Figure 2.2 with the line segments a,b,c, and d. The first difficulty is how to compare them? If we consider the y-projections of their left endpoints as in Figure 2.1 we obtain the ordering a,c,b,d, but if we take the y-projections of their right endpoints we obtain c,b,d,a. This reflects the well known fact that, in general, non-parallel line segments in the plane cannot be totally ordered. Condition (2) stems from the need for a semi-dynamic search structure to be organized according to a fixed total order. Since non-isothetic polygons do not satisfy Condition (2) it appears that semi-dynamic structures cannot be obtained in this setting. This, fortunately, is not the case, since we can partition a set of polygons in the plane into *maximal zig-zags*, which are a restricted class of line segment curves. The zig-zags, as we shall see, are totally ordered just as the endpoints and horizontal line segments are in the isothetic case. Thus for these more complex elements Condition (2) is satisfied and semi-dynamic search structures can be used. Indeed the same semi-dynamic

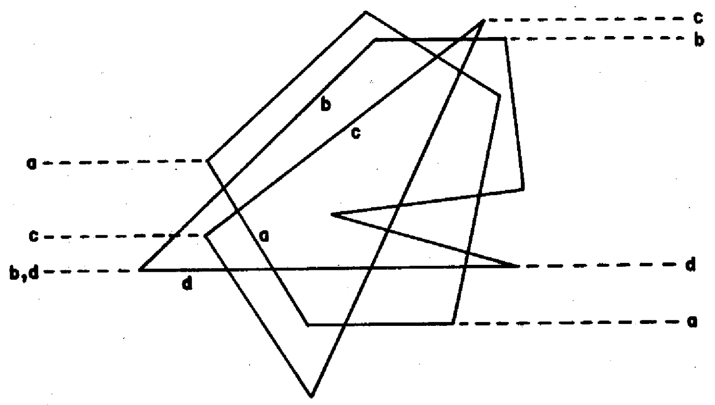

Figure 2.2

search structures that are introduced in [BW, E, Mc, VW, Wo] for isothetic problems can be used, directly, for their non-isothetic variants. This translation result is demonstrated in the present paper for the visibility tree [Wo] and the contour problem. Before we present the algorithm we must first define the *zig-zag partition* of a set of polygons. For this purpose we identify polygons with their edges. Moreover we assume no vertical edges are present and no three edges have a common intersection point.

Definition A *zig-zag* is a line segment curve which is monotonic with respect to x. Given a set S of polygons in the plane a zig-zag in S is *maximal* if it cannot be extended at either end. Two zig-zags z and z' in S *cross* if there are points (x_1,y_1) and (x_2,y_2) in z and points (x_1,y_1') and (x_2,y_2') in z' such that $y_1 < y_1'$ and $y_2 > y_2'$. The *zig-zag partition* of S is a set Z of maximal zig-zags satisfying $\bigcup_{z \in Z} edges(z) = \bigcup_{s \in S} edges(s)$ and for all z_1, z_2 in Z, $z_1 \neq z_2$ implies z_1 and z_2 do not cross and $z_1 \cap z_2$ is a finite set of points. Such a partition is unique and consists of $O(n)$ maximal zig-zags. Note that if two zig-zags z_1 and z_2 in the partition share an x-value x, that is (x,y_1) is in z_1, (x,y_2) is in z_2 and $y_1 < y_2$, then z_1 is *below* z_2 at all shared x-values.

In Figure 2.4 the zig-zag partition of the polygons of Figure 2.2 is displayed. Algorithmically a maximal zig-zag can be found by starting at the leftmost point p of a polygon and tracing out one of the paths from p following the rules illustrated in Figure 2.3. The transitive closure of the *below* relation induces a partial order in Z. This is extended to a total order by also defining z_1 to be below z_2 if they share no x-value and z_1 is to the left of z_2. This ordering was introduced in [GY] and is called the 'below plus consulting left' relation there. The total order of the zig-zag partition in Figure 2.4 is given at the right of the polygons. In Figure 2.5 we display the zig-zag partition of a standard set of isothetic polygons rotated through 45°.

Using the Bentley-Ottmann-Shamos-Hoey [BO] algorithm with Brown's

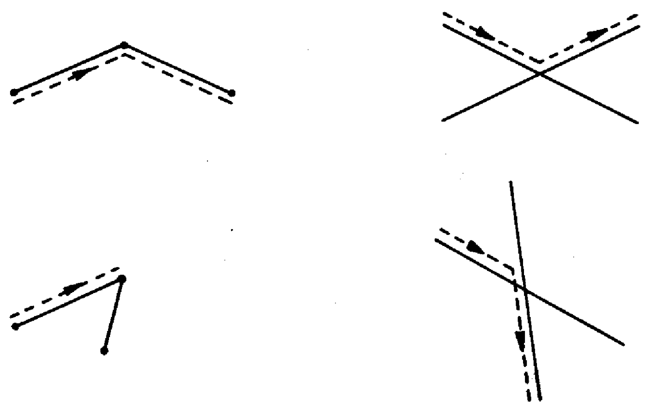

Figure 2.3

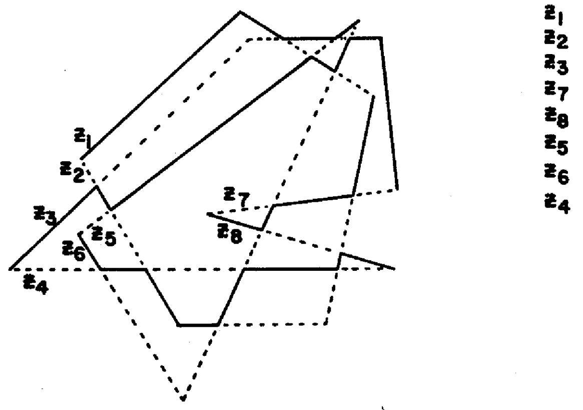

Figure 2.4

improvement [Br] the zig-zag partition can be computed in $O((n+k)\log n)$ time and $O(n)$ space [OW1], where n is the total number of vertices in the polygons and k is the number of pair-wise edge-intersections. Even if this can be improved to require $O(n \log^2 n + k)$ time along the lines of [C], because the contour algorithm requires $O((n+k)\log n)$ time and $O(n)$ space, there is little incentive for the attempt.

In the next section we derive the contour algorithm.

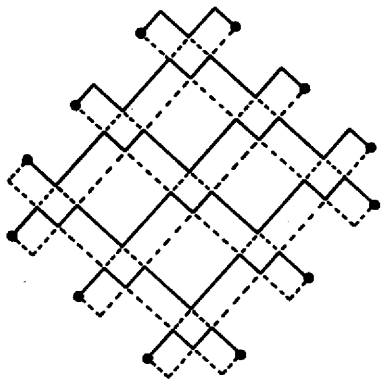

Figure 2.5

3. THE CONTOUR ALGORITHM

Before giving a high-level version of the *CONTOUR* algorithm we introduce some additional useful terminology.

Each zig-zag has one left endpoint and one right endpoint. Each sweep point within a zig-zag is either a *bend* point, that is a common endpoint of two edges, or an *intersection* point. These are displayed in Figures 3.1(a)-(d); the interior of the polygon is indicated by shading.

During the plane-sweep of the polygons the edges of the zig-zags (or the polygons) can be classified as being either *blocking* or *unblocking* edges. An edge is blocking if the interior of its polygon appears to its right, Figure 3.2(a), and is unblocking otherwise, Figure 3.2(b).

During a plane sweep bend points do not affect the contour except to introduce a new vertex. However endpoints and intersection points may affect the contour as a careful examination of Figure 3.1 indicates. In extending the isothetic contour algorithm to the non-isothetic case we treat endpoints as if they are vertical edges. Bend points are treated as if they correspond to the common endpoint of two horizontal edges, while intersection points are treated similarly to either endpoints or bend points. In Figure 3.1(d) the first and fourth intersection points are similar to bend points, and the second and third to endpoints.

Following [Wo] and replacing references to horizontal and vertical line segments with zig-zags we have the following high-level algorithm.

Algorithm *CONTOUR*

On entry: A set S of p simple polygons which have no vertical edges and in which no three edges have a common intersection point.

On exit: The edges of the union of S such that no two adjacent edges have the same direction.

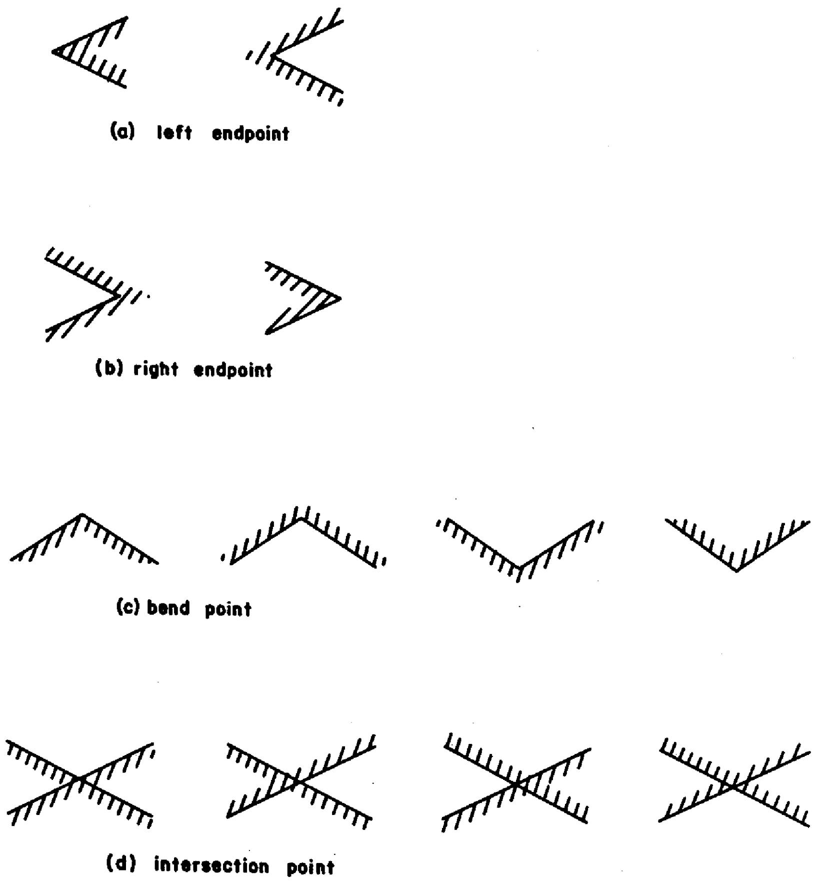

Figure 3.1

```
begin

1.  Call the zig-zag partition algorithm [OW1] — this yields the $O(n+k)$ sweep
    points in $x$-sorted order and the $O(n)$ zig-zags in *below* order, that is in
    their total ordering. Initialize the set $V$ of visible active zig-zags to $\emptyset$ and
    the set $B$ of blocking intervals to $\emptyset$.

2.  For each sweep point $x$, corresponding to a left, right, bend, or intersection
    point do

    2.1  $x$ is a left endpoint.
         Activate the corresponding two zig-zags. If either or both of them are
         visible add them to $V$ and initiate the corresponding edges of the con-
         tour. Add their open interval to $B$ and if this blocks any previously
         visible zig-zags then remove them from $V$ and terminate the
         corresponding edges.
```

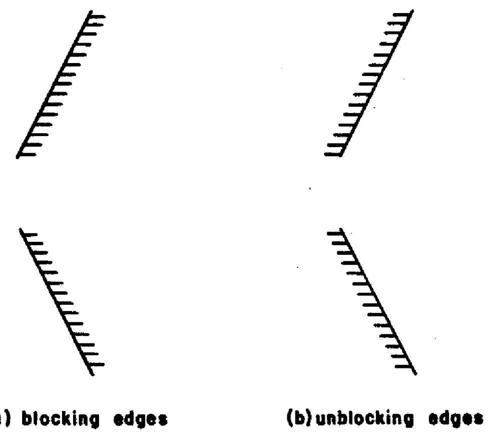

Figure 3.2

```
2.2 $x$ is a right endpoint.
    De-activate the corresponding two zig-zags. If either or both of them
    are visible remove them from $V$ and terminate the corresponding edges
    of the contour. Remove their open interval from $B$ and if this
    unblocks any previously blocked zig-zags then add them to $V$ and ini-
    tiate the corresponding edges.
2.3 $x$ is a bend point.
    Update the current position in the zig-zag corresponding to $x$.
2.4 $x$ is an intersection point.
    Update the current positions in the two zig-zags corresponding to $x$,
    removing them from $V$ and terminating their visible edges if necessary
    (see Figure 3.1(d)).
end CONTOUR.
```

Step 1 implies that CONTOUR requires $O(n+k)$ space since the zig-zags are pre-computed and available for Step 2. But this is not necessary, it is sufficient only to determine the left endpoint of each zig-zag and their total order during Step 1. This then requires only $O(n)$ space, since there are only $O(n)$ zig-zags. Additionally during Step 2 the re-discovery of the zig-zags is carried on as a background task using the algorithm in [OW1], again with an $O(n)$ space requirement. For simplicity of presentation however we assume that the zig-zags are pre-computed in Step 1.

Step 2 requires a data structure for B and V which supports:

(i) the insertion of a zig-zag into V, the determination of its visibility status, and if it is visible initiating the corresponding visible edge.

(ii) the deletion of a zig-zag from V, the determination of its visibility status, and if it is visible terminating the corresponding visible edge.

(iii) the insertion of an open interval into B, terminating the edges of all newly hidden zig-zags.

(iv) the deletion of an open interval from B, initiating the edges of all newly visible zig-zags, and releasing the blocking intervals or portions thereof, which belong to the same polygon and which it overlaps.

In [Wo] the *visibility tree* is introduced for the isothetic case. We demonstrate that it also fills our present needs. It requires $O(n)$ space, $O(\log n)$ time for each of (i) and (ii), $O(\log n + e)$ time for (iii), where e is the number of terminated edges, and $O(r \log n + e)$ time for (iv), where r is the number of released intervals and e the number of initiated edges.

The visibility tree is constructed as follows. Let there be m zig-zags which are enumerated as $z_1, \ldots, z_m$ so that z_i *below* z_j if and only if $i < j$. Let T be a minimal-height binary tree with m external nodes, labelled in left-to-right order with $z_1, \ldots, z_m$. The external node labelled z_i also represents the closed-open interval $[z_i, z_{i+1})$, $1 \leq i \leq m$, where $z_{m+1} = +\infty$. Such an interval is well defined since zig-zags never cross each other. We call such an interval a *zig-zag interval* or *z-interval* for short. In a natural manner each internal node u represents a closed-open interval $[z_l, z_{r+1})$, where z_l is the lowest zig-zag, with respect to *below*, in its subtree $T(u)$ and z_r is the highest, with respect to *below*. Thus the *root* of T represents the interval $[z_1, +\infty)$.

With each node u in T we associate the following values:

(i) $interval(u)$ — the interval represented by u.

(ii) $active(u)$ — for external nodes only, whether the associated zig-zag is currently active or not.

(iii) $\#cover(u)$ — the number of z-intervals which cover or block u, see below.

(iv) $visible(u)$ — the set of active zig-zags in $T(u)$ that are visible if $T(u)$ is considered independently from T.

Initially, for each node u in T, $active(u)$ is **false**, $\#cover(u) = 0$, and $visible(u) = \emptyset$. Note that for the sets B and V, B is the set of z-intervals in T without their bottommost endpoint and $V = visible(root)$. Let $I = (z_i, z_j)$, $1 \leq i < j \leq m$, be an open z-interval, then I is inserted into T at all nodes u in T which satisfy:

$$interval(u) \subseteq [z_i, z_j) \qquad \text{and} \qquad interval(parent(u)) \not\subseteq [z_i, z_j) \ ,$$

where $parent(u)$ has the obvious meaning. Inserting the closed-open interval corresponding to an open interval causes some minor technical difficulties which we treat below. Each such node u is said to be *covered* by I. It can be shown that at most $O(\log m)$ nodes are covered by any such I, for example see [BW].

Moreover at every covered node u we add one to *#cover*(u). Also *active*(u) will be set of **true**, where u has z-value z_i or z_j, if the zig-zags are initiating edges. If they are terminating edges then *active*(u) will be set to **false**.

In Figure 3.3 we display an example visibility tree in which we only indicate whether or not *#cover*$(u) = 0$. Two z-intervals have been inserted: (z_1, z_5) and (z_3, z_5); the covered nodes are filled in. There are 6 active points, z_1, z_2, z_3, z_5, z_6, and z_7; these are indicated by check marks. The visible field of each node is displayed. Observe that the removal of (z_3, z_5) does not make either z_3 or z_5 visible, since they are still blocked by (z_1, z_5).

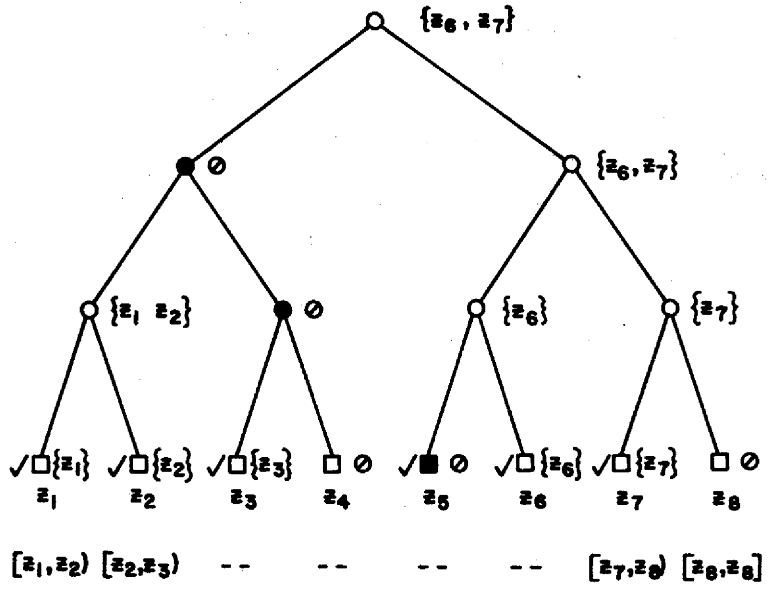

Figure 3.3

The cover-search algorithm for such a tree is as follows:

Algorithm *COVERSEARCH* (T, u, I)

On entry: A visibility tree T with root u, and a z-interval $I = [z_i, z_j)$, $1 \le i < j \le m$.

On exit: The nodes covered by I are given.

```
begin
    if $u$ is an external node then
        $\{interval(u) \cap I \neq \emptyset\}$ report $u$;
        return
    else {$u$ is an internal node }
    if $interval(u) \subseteq I$ then
        report $u$;
```

```
            return
    else {$interval(u) \not\subseteq I$}
    begin
        if $interval(left(u)) \cap I \neq \varnothing$ then
                $COVIRSEARCH(T, left(u), E)$;
        if $interval(right(u)) \cap I \neq \varnothing$ then
                $COVERSEARCH(T, right(u), I)$;
            return;
    end
end COVERSEARCH.
```

It is important to note that $visible(u)$ can be reconstructed from the visible sets at u's children. In other words:

```
    if $\#cover(u) = 0$ then
        $visible(u) := visible(left(u)) \cup visible(right(u))$ ;
            else
        $visible(u) := \varnothing$
```

where $left(u)$ and $right(u)$ have the obvious meanings.

Whenever a z-interval I is inserted into or deleted from T we update the *visible* sets for all nodes which I covers and for all ancestors of these nodes using the reconstruction rule given above. Fortunately I not only covers at most $O(\log m)$ nodes, but these nodes also have a total of $O(\log m)$ ancestors. Of course this updating need only be carried out when $\#cover(u)$ becomes either zero or non-zero. Unfortunately forming the union of two visible sets may take $O(n)$ time, but we discuss below how this can be avoided.

It only remains to discover the newly hidden and newly visible zig-zags on inserting a z-interval I into T or deleting a z-interval I from T, respectively. Assume I, corresponding to two zig-zags, covers node u, $\#cover(u) = 0$, and $visible(u) \neq \varnothing$. Then not only do we obtain $\#cover(u) = 1$, but also $visible(u)$ becomes $\varnothing$, since all zig-zags in $visible(u)$ are now blocked. However they may not be newly hidden, since some ancestor of u may be covered already by some other edge. Fortunately during *COVERSEARCH* we can keep track of the cover status of the ancestors of the current node. We introduce a boolean parameter *upcover* for this purpose. The modification of *COVERSEARCH* to give $COVERSEARCH(T, u, I, upcover)$ is easily obtained.

To summarize, if a z-interval I covers a node u, with $\#cover(u) = 0$ and $visible(u) \neq \varnothing$, then:

```
    Set $\#cover(u)$ to 1 ;
    if not upcover then terminate all edges corresponding to zig-zags in $visible(u)$ ;
    Set $visible(u)$ to $\varnothing$ .
```

This deals with the termination of edges. We now consider their initiation.

When deleting a z-interval I of a polygon Q, we need to determine the z-intervals of Q which are released by I. Now although many z-intervals can be released, it cannot release 'territory' which it doesn't own. This is illustrated in Figure 3.4, where on the left is a sequence of z-intervals of Q at the current scan point. The relationship of a z-interval I to the z-intervals of Q at the current scan point can only be of the forms shown in Figure 3.4 as z-intervals (a)-(d). We cannot have the z-interval (e) of Figure 3.4 since this would involve releasing an unblocked z-interval, corresponding to a cavity or hole Q .

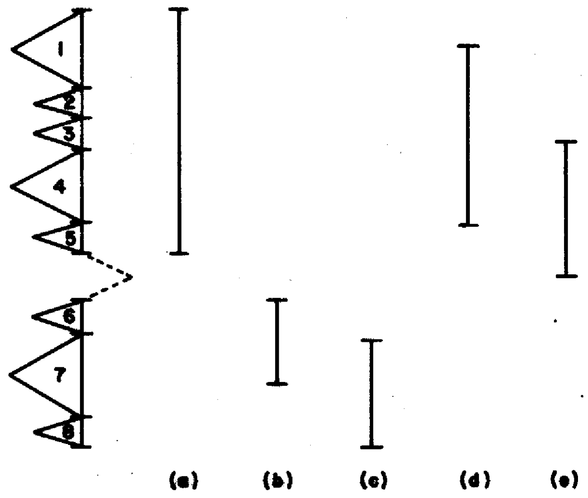

Figure 3.4

To find the z-intervals to be released by I it suffices to determine which z-intervals belonging to Q are stabbed by its endpoints, remove all z-intervals belonging to Q in between and modify the stabbed z-intervals appropriately. For example, in Figure 3.4(d) I stabs z-intervals 1 and 4, causing 2 and 3 to be completely released and 1 and 4 to be partly released. Since z-intervals in the visibility tree do not identify the polygon they belong to, we use a subsidiary balanced search tree, for each polygon, for this purpose. Initially these trees are empty, but whenever a z-interval of a polygon Q is inserted into the visibility tree, it is also inserted into Q's search tree, or rather its endpoints are. The determination of the z-intervals to be deleted from a search tree and the visibility tree on meeting a z-interval, as in Figure 3.4(d), is now straightforward. Assume the z-interval belongs to polygon Q and it corresponds to (z_i, z_j). Then Q's search tree is queried with both z_i and z_j , separately. They are either both found or one or both of them stab a z-interval internally. To find the endpoints of the stabbed intervals, simply carry out a successor or predecessor search. In all cases two z-values z'_i and z'_j are found in the search tree satisfying $z'_i \leq z_i < z_j \leq z'_j$. All values from z'_i to z'_j in the search tree are deleted from it and, moreover, the z-intervals they determine are deleted from the visibility tree. If $z'_i \neq z_i$ then (z'_i, z_i) is a z-interval and, similarly, if $z'_j \neq z_j$ then (z_j, z'_j) is a z-interval. If either or both of these z-intervals exist they are inserted into Q's search tree and into the visibility tree. Thus we are only left with the problem of how the previously hidden zig-zags which start newly visible edges are found.

Consider a node u which is covered by I, a released z-interval. Then we can recompute $visible(u)$ from its children. $Visible(u)$ contains, however, only candidate visible zig-zags or edges, since they may still be blocked further up the tree. But once more we can make use of parameter *upcover* to discover whether or not this is the case, as in insertion.

We have, in the foregoing, concentrated exclusively on actions (iii) and (iv) on the visibility tree, so we now turn, belatedly, to actions (i) and (ii), the insertion and deletion of a zig-zag z. In both cases we search for the external node u representing z in the tree. If z is inserted then $active(u) := \mathbf{true}$ and $visible(u) := \{z\}$ if $cover(u) = \varnothing$. If z is deleted then $active(u) := \mathbf{false}$ and $visible(u) := \varnothing$. In both cases the visible sets are updated for all nodes on the search path, and their corresponding edges are initiated or terminated if necessary.

Efficiency Considerations

To avoid copying of visible sets during a disjoint-union operation we keep only one global copy of the visible set associated with the root of the visibility tree. This is represented as a doubly-linked list, denoted by *GV*. Initially *GV* is empty. At each node u in the tree $visible(u)$ is represented by two pointers $firstv(u)$ and $lastv(u)$. They either both point to the first and last elements of $visible(u)$ in *GV* or both are *nil*. The reason for this is that $visible(u)$ is the set of zig-zags visible with respect to $T(u)$, however they may be blocked further up the tree and, therefore be hidden globally.

With this representation the operation:

$$visible(u) := visible(left(u)) \cup visible(right(u))$$

can be carried out as the simple catenation of two doubly-linked lists, that is the elements of the doubly-linked list pointed to by $lastv(left(u))$ and $firstv(right(u))$ should be linked together while

$$firstv(u) := firstv(left(u))$$

and

$$lastv(u) := lastv(right(u)).$$

Note that the order of the elements in $visible(left(u))$ and in $visible(right(u))$ is not disturbed by their catenation. Thus the visible and hidden sets at nodes in $T(u)$ are unaffected by this operation.

We have replaced a putatively $O(n)$ time union with a constant time union and also reduced the space requirements for $visible(u)$ to a constant rather than $O(n)$.

It is now straightforward to determine that actions (i) and (ii) require $O(\log m)$ time. This follows because a search requires $O(\log m)$ time, the updating at the corresponding external node requires constant time, and recomputing a visible set at an ancestor also requires constant time (the catenation of two lists), thus $O(\log m)$ time over all.

For action (iii) *COVERSEARCH* requires $O(\log m)$ time to determine the nodes to be updated, the updating of the visible sets at these nodes requires $O(\log m)$ time, and the initiation or termination of e edges requires $O(e)$ time, giving $O(\log m + e)$ time overall. Now an action of type (iv), that is a single deletion of a z-interval corresponding to a right edge can cause $O(n)$ z-intervals to be released from the visibility tree, that is it can take $O(n \log m)$ time. However the time for deleting $n+k$ z-intervals is $O((n+k)\log m)$ overall. In other words $O(\log m)$ time when amortized over the $n+k$ deletion operations. To see this use a charging argument. When a deletion causes a z-interval to be completely released charge the z-interval one unit. When a deletion causes a z-interval to only be partly released, then charge the deletion operation one unit. Each deletion may partly release at most two z-intervals, hence the total charged to the at most $n+k$ deletions is $2n+2k$. Similarly each z-interval can only be charged once, when it is finally released completely. This results in a total charge of at most $n+k$, giving a grand total of at most $3(n+k)$ units. Each unit represents a constant number of deletions and possible insertions, that is $O(\log m)$ time. Thus, overall, the deletions take $O((n+k)\log m)$ time.

The space requirements for the visibility tree during *CONTOUR* is $O(n)$ as is the space requirement for the p subsidiary search trees, since there are at most n distinct zig-zags. Moreover the visible sets also require $O(n)$ space, since each of the n zig-zags appears at most once in the global visibility list, and constant space is required at each node. Thus the tree requires $O(n)$ space in the worst case since $2 \le m \le n$.

These remarks lead to:

Theorem 3.1 *Given a set of simple polygons consisting of* n *vertices, the edges of the resulting disjoint polygons can be computed in* $O((n+k)\log n)$ *time and* $O(n)$ *space, where* k *is the number of edge intersections.*

Finally, if the given polygons are convex, then the space requirement can be reduced since we only keep one z-interval for each polygon. Thus we obtain:

Theorem 3.2 *Given a set of* p *convex polygons consisting of* n *vertices, the edges of the resulting disjoint polygons can be computed in* $O((n+k)\log n)$ *time and* $O(p)$ *space, where* k *is the number of edge intersections.*

REFERENCES

[BO] Bentley, J.L. and Ottmann, Th., Algorithms for Reporting and Counting Geometric Intersections, *IEEE Transactions on Computers C-28* (1979), 643-647.

[BW] Bentley, J.L., and Wood, D., An Optimal Worst-Case Algorithm for Reporting Intersections of Rectangles, *IEEE Transactions on Computers C-29* (1980), 571-577.

[Br] Brown, K.Q., Comments on 'Algorithms for Reporting and Counting Geometric Intersections', *IEEE Transactions on Computers C-29* (1981), 147-148.

[C] Chazelle, B., Reporting and Counting Arbitrary Planar Intersections, Brown University, Computer Science Technical Report CS-83-16, 1983.

[E] Edelsbrunner, H., Dynamic Data Structures for Orthogonal Intersection Queries, Technical University of Graz, Graz, Austria, Institute für Informationsverarbeitung, Report F59 (1980).

[G1] Güting, R.H., An Optimal Contour Algorithm for Iso-Oriented Rectangles, *Journal of Algorithms* (1984), to appear.

[G2] Güting, R.H., Optimal Divide-and-Conquer Algorithms to Compute Measure and Contour for a Set of Iso-Rectangles, *Acta Informatica* (1984), to appear.

[G3] Güting, R.H., Conquering Contours: Efficient Algorithms for Computational Geometry. Doctoral Dissertation, Universität Dortmund, 1983.

[GY] Guibas, L.J., and Yao, F.F., On Translating a Set of Rectangles, *Proceedings of the 12th Annual ACM Symposium on Theory of Computing* (1980), 154-160.

[LP] Lipski, W., and Preparata, F.P., Finding the Contour of a Union of Iso-Oriented Rectangles, *Journal of Algorithms 1* (1980), 235-246.

[Mc] McCreight, E.M., Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles, Report CSL-80-9, Xerox PARC, (1980).

[NP] Nievergelt, J. and Preparata, F.P., Plane-Sweep Algorithms for Intersecting Geometric Figures, *Communications of the ACM 25* (1982), 739-747.

[OW1] Ottmann, Th. and Widmayer, P., On Translating a Set of Line Segments, *Computer Vision, Graphics, and Image Processing 24* (1983), 382-389.

[OW2] Ottmann, Th., and Widmayer, P., Solving Visibility Problems by Using Skeleton Structures. *Mathematical Foundations of Computer Science, Springer-Verlag Lecture Notes in Computer Science 176* (1984), 459-470.

[VW] Vaishnavi, V.K., and Wood, D., Rectilinear Line Segment Intersection, Layered Segment Trees and Dynamization, *Journal of Algorithms 2* (1982), 160-176.

[W] Widmayer, P., *Computational Complexity in Computer Graphics and VLSI Design.* Doctoral Dissertation, Universität Karlsruhe, 1983.

[Wo] Wood, D., The Contour Problem for Rectilinear Polygons, *Information Processing Letters* (1984), to appear.