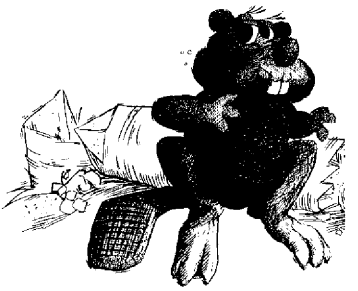
UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT



The Contour Problem for Rectilinear Polygons

Derick Wood

Data Structuring Group
CS-83-20
(Revised)

October, 1983

UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

THE CONTOUR PROBLEM FOR RECTILINEAR POLYGONS †

Derick Wood (1)

ABSTRACT

The union of a set of p, not necessarily disjoint, rectilinear polygons in the plane determines a set of disjoint rectilinear polygons. We present an $O(n \log n)$ time and $O(n)$ space algorithm to compute the edges of the disjoint polygons, that is the contour, where n is the total number of edges in the original polygons and r the total number in the resulting set. This time- and space-optimal algorithm uses the scan-line paradigm as in two previous approaches to this problem for rectangles, but requires a simpler data structure. Moreover if the given rectilinear polygons are rectilinear convex the space requirement is reduced to $O(p)$.

1. INTRODUCTION

The contour problem was introduced by Lipski and Preparata [LP] for a set of rectilinear-oriented rectangles. The union of such a set determines a set of rectilinear polygons or *r-polygons* as we call them. The contour problem is to produce the usual description of these r-polygons, that is as contour cycles. In [LP] this objective is achieved in two stages. In the first stage an algorithm to compute the edges of the resulting r-polygons is given which requires $O(n \log n + n \log (n^2/r))$ time and $O(n)$ space. In the second stage the descriptions of each resulting r-polygon is produced in a total of $O(r)$ time and $O(r)$ space.

Clearly the algorithm in the first stage is not worst-case optimal. Gülting [G1, G3] was able to design a worst-case-time optimal algorithm for the first stage, which is based upon the same ideas as [LP] but introduces a more efficient data structure, the *contracted segment tree*. Later in [G2, G3] he was also able to find a time-optimal divide and conquer algorithm for the same

---

† Work supported by a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692.

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

problem.

Our aim in the present paper is two-fold.

First, we claim that the generalization of the problem to r-polygons, although minor, gives a unifying framework for the problem. Both the input and output data are sets of r-polygons. Moreover r-polygons have recently become the subject of intensive investigation, since they occur in image processing and VLSI design for example, see [MF] and [NLLW].

Second, we also describe an algorithm for the first stage, which is worst-case time- and space-optimal and uses the scan-line paradigm, but uses a data structure, *the visibility tree*, which is a simpler variant of the segment tree than the contracted segment tree.

In Section 2 we present a high-level approach to the contour problem, while in Section 3 we present the modified segment tree, the visibility tree, which we use as our central data structure. This provides us with an $O(n \log n + r)$ time and $O(n)$ space algorithm for a set of r-polygons with n edges resulting in a set of disjoint r-polygons with r edges. Moreover when given p r-convex r-polygons the space requirement is reduced to $O(p)$.

An *r-polygon* (rectilinearly-oriented polygon) is a polygon with edges composed of horizontal and vertical line segments (r-line segments). An r-polygon P is *r-convex* if the intersection of P with any horizontal or vertical line is either empty or a connected set.

2. BREAKING THE PROBLEM DOWN

In this section we introduce our approach to solving the contour problem, and then present in the following section the necessary data structure.

Let P be the initial set of p r-polygons. Then we have the decomposition of the problem into two stages.

Stage 1: Compute the horizontal edges of the set of resulting r-polygons, R.

Stage 2: Deduce the corresponding vertical edges and output the description of each r-polygon in edge-cycle form.

We assume that Stage 2 is dealt with as in [LP]. We solve Stage 1 by using the scan-line paradigm, that is we sweep, conceptually, a vertical line from $x = -\infty$ to $x = +\infty$, and as it sweeps over P we produce the horizontal edges of R, as we sweep along them, see Figure 2.1.

The key observations about the scan-line approach are that, in reality, we only need to sweep from one vertex to the next one (in sorted x-order), and that we only need to keep track of the horizontal edges in P which currently intersect the scan line, the *active* edges, see Figure 2.1.

However the problem is not as simple as this would lead us to believe,

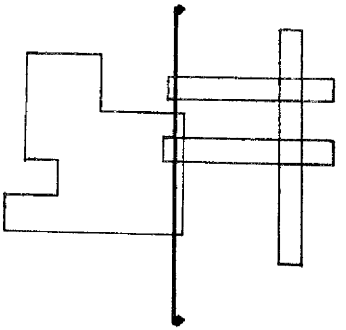


Figure 2.1

since horizontal edges in P may never or may often contribute edges to R. In other words we have to ascertain whether or not an active edge is currently visible or hidden with respect to R, cf. [OWW]. Now an edge can only change its visibility status when a vertical edge is met. There are two kinds of vertical edges, left and right, see Figure 2.2.

Figure 2.2

On meeting a left vertical edge, Figure 2.2(a), horizontal edges which intersect it become hidden if they were not already hidden, and remain hidden otherwise. On meeting a right vertical edge, Figure 2.2(b), horizontal edges which intersect it may become visible. In this case their visibility status depends upon whether or not they are hidden or blocked by previous left edges of other active r-polygons.

The crucial idea behind our approach is the separation of concerns in maintaining the visibility status of horizontal edges. First, active horizontal edges are initiated and terminated on meeting vertical edges. Thus we sweep through the x-sorted vertical edges rather than the vertices of the r-polygons. Second, active horizontal edges need only be represented by points on the scan line. Third, we partition the active horizontal edges into an H, for hidden, set and a V, for visible, set, and these sets are maintained during the scanning. Finally, vertical edges also act as "blockers" and "unblockers" as far as the visibility status of active edges is concerned.

Algorithm *CONTOUR*

On entry we have a set of p r-polygons P. On exit we have the r directed horizontal edges of R, the union of P.

begin

1. Sort the vertical edges in P into ascending x-order, the corresponding x-values we call *scan points*. Initialize two sets H and V to $\varnothing$, the hidden and visible sets of horizontal edges.
2. For each scan point x, corresponding to a vertical edge E and r-polygon Q, in turn do:

   The endpoints of E may initiate or terminate horizontal edges. Those which may initiate an edge are used to update H or V accordingly.

   2.1 E is a left vertical edge.
   Determine all edges in V which are blocked and add these to H, removing them from V, and terminating the horizontal edges they represent, except for any edge in V which belongs to Q.

   2.2 E is a right vertical edge.
   Determine all edges in H which are unblocked and also become visible. Add them to V, initiating the horizontal edges they represent, and removing them from H.

   Finally the endpoints of E which may terminate an edge are used to update H or V accordingly.

end *CONTOUR*.

A number of details have been ignored in this high-level description, some of which we now discuss.

When sorting vertical edges in Step 1, if two vertical edges have the same x-projection we always place left edges before right edges in the ordering, this prevents the production of horizontal edges of measure 0.

The left vertical edges come in four varieties, see Figure 2.3, as do right vertical edges. For example the left edge of Figure 2.3(a) initiates two horizontal edges, while that of Figure 2.3(d) initiates one and terminates one. This distinction is made in Step 2 of our algorithm. Although directions are not mentioned explicitly in the body of the algorithm, we assume that the representation of active horizontal edges includes this information.

Although blocking an active horizontal edge may be done simply by marking it blocked, unblocking requires more than this binary information. We could assume each active horizontal edge has a blocking count which is then reduced by one on meeting a unblocking edge. However a little thought shows that the initialization of such a value remains a non-trivial task. The

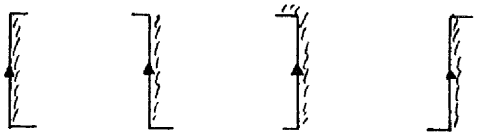


Figure 2.3

data structure we introduce solves this problem by maintaining information about left edges of r-polygons, which have not yet been released.

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

(i) the insertion of a point, corresponding to a horizontal edge, the determination of its visibility status, and if it is visible initiating a visible edge.

(ii) the deletion of a point, corresponding to a horizontal edge, the determination of its visibility status, and if it is visible terminating a visible edge.

(iii) the insertion of a left vertical edge, terminating all newly hidden edges.

(iv) the deletion of a right vertical edge, initiating all newly visible edges.

Moreover the data structure should take $O(\log n)$ time for actions (i) and (ii), and $O(\log n+r)$ time for actions (iii) and (iv), where r is the number of edges affected.

In the next section we describe such a data structure.

3. THE VISIBILITY TREE

The data structure, which we call the visibility tree, is basically a segment tree, for example see [BW] and [LP], with additional information represented at its nodes. We first give a conceptual description of the visibility tree, before explaining an efficient representation for it.

The endpoints of the n edges of the r-polygons in P provide at most n distinct y-coordinate values, $y_1 < y_2 < \cdots < y_m$. These divide the y-axis into $m+1$ intervals or *fragments*. Letting T be a minimal-height binary tree with m leaves, we label the leaves from left to right with $y_1, \cdots, y_m$.

The leaf labelled with y_i also represents the closed-open interval $[y_i, y_{i+1})$, $1 \leq i < m$, and the m-th leaf represents $[y_m, y_m]$. In a natural manner each internal node u either represents a closed-open interval $[y_l, y_r)$, where y_l is the leftmost leaf in its subtree, and y_r rightmost, or represents $[y_l, y_m]$ if y_m is the rightmost leaf in u's subtree. Thus the root of T represents the interval $[y_1, y_m]$.

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

(i) *interval*(u), the interval represented by u.
(ii) *hits*(u), for leaves u only, the number of times the leaf value has been inserted and not deleted.
(iii) *cover*(u), the set of r-polygons which cover u, see below.
(iv) *visible*(u), the set of active points in $T(u)$ that are visible with respect to $T(u)$.
(v) *hidden*(u), the set of active points in $T(u)$ that are hidden with respect to $T(u)$.

Initially, *cover*(u) = *visible*(u) = *hidden*$(u) = \varnothing$, and at any later time *visible*$(u) \cap$ *hidden*$(u) = \varnothing$, for all nodes in u. Note that for the sets V and H of Section 2 $V =$ *visible*(*root*) and $H =$ *hidden*(*root*). Let $E = [y_i, y_j]$, $1 \leq i < j \leq m$, be a left vertical edge of some r-polygon Q, then E is inserted into T at all nodes u in T which satisfy:

$$\textit{interval}(u) \subseteq [y_i, y_j] \quad \text{and} \quad \textit{interval}(\textit{parent}(u)) \not\subseteq [y_i, y_j] ,$$

where *parent*(u) has the obvious meaning. Each such node u is said to be *covered* by E or by Q. It can be shown that at most $O(\log n)$ nodes are covered by any such E, for example see [BW]. Moreover at every covered node u we add E, and hence Q, to *cover*(u). Without more ado we give the cover-search algorithm for such a tree:

Algorithm *COVERSEARCH* (T, u, E, Q)

On entry a visibility tree T with root u, and an edge $E = [y_i, y_j]$, $1 \leq i < j \leq m$, of r-polygon Q. On exit the nodes covered by E are given.

```
begin
    if u is a leaf then
        if interval(u) ⊆ E then report u;
        return
    else {u is not a leaf }
    if interval(u) ⊆ E then
            report u;
            return
    else {⊄ E}
    begin
        if interval(left(u)) ∩ E ≠ ∅ then
                COVERSEARCH(T, left(u), E, Q);
        if interval(right(u)) ∩E ≠ ∅ then
                COVERSEARCH(T,right(u), E, Q) ;
            return;
    end
end COVERSEARCH.
```

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

```
if cover(u) = ∅ then
    visible(u) := visible(left(u)) ∪ visible(right(u)) ;
    hidden(u) := hidden(left(u)) ∪ hidden(right(u))
        else
    visible(u) := ∅
    hidden(u) := hidden(left(u)) ∪ hidden(right(u))
                    ∪ visible(left(u)) ∪ visible(right(u)) ,
```

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

Whenever vertical edge E is inserted into or deleted from T we update the *visible* and *hidden* sets for all nodes which E covers and for all ancestors of these nodes. Fortunately E not only covers at most $O(\log m)$ nodes, but also these nodes have at most $O(\log m)$ ancestors. Of course this updating need only be carried out when *cover*(u) becomes either empty or non-empty. Now forming the union of two sets may take $O(n)$ time, but we discuss below how to avoid this.

It only remains to discover the newly hidden and newly visible points on inserting a left edge E into T or deleting a right edge E from T, respectively. Assume E, a left edge, covers node u, *cover*$(u) = \emptyset$, and *visible*$(u) \neq \emptyset$. Then not only do we obtain *cover*$(u) = \{E\}$, but also *visible*(u) becomes $\emptyset$, since all points in *visible*(u) are now hidden. 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 every covered node, and therefore also report this. We introduce a boolean parameter *upcover* for this purpose. The modification of *COVERSEARCH*$(T,u,E,Q,upcover)$ is left to the interested reader. Summarizing we have for a left edge E which covers a

node u, with $cover(u) = \emptyset$ and $visible(u) \neq \emptyset$:

Set $cover(u)$ to $\{E\}$;
if not upcover **then** terminate all edges in $visible(u)$;
Set $hidden(u)$ to $hidden(u) \cup visible(u)$;
Set $visible(u)$ to $\emptyset$.

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

On meeting a right edge E of an r-polygon Q, we need to determine the left edges of Q or portions thereof which are released by E. Now although many edges of Q can be released, we cannot release 'territory' which we don't own, that is the relationship of a right edge of Q to the remaining portions of its left edges at the current scan point can only be of the forms shown in Figure 3.1 (a)-(d). We cannot have Figure 3.1 (e) since this would involve releasing an unblocked interval, corresponding to a cavity or hole in Q .

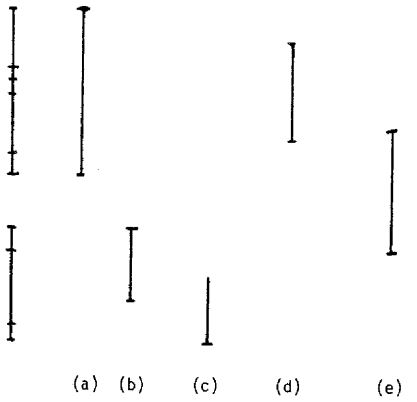


Figure 3.1

To find the edges to be released by a right edge of Q it suffices to determine which left edges are stabbed by its endpoints, remove all left edges in between and modify the stabbed edges appropriately. To find all edges in between efficiently, it suffices to keep with each left edge E (or portion thereof) of Q the predecessor and successor adjacent left edge (the ones having an endpoint in common with E) of Q in T. These are then deleted from T. Dealing with the two, at most, stabbed edges is also straightforward, first delete them and then re-insert their remnants. Thus we only have to

consider how we determine the points which become the starting points of newly visible edges.

Consider a node u which is covered by E a released left edge of Q. Then we can recompute *visible*(u) from its children. *Visible*(u) contains, however, only candidate visible 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 may be 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 point y, that is a horizontal edge. In both cases we search for the leaf u representing y in the tree. If y is inserted then *hits*$(u) :=$ *hits*$(u)+1$ and if *hits*(u) becomes non-zero then *visible*$(u) := \{y\}$ if *cover*$(u) = \emptyset$ and *hidden*$(u) := \{y\}$ otherwise. If y is deleted then *hits*$(u) :=$ *hits*$(u)-1$ and if *hits*(u) becomes zero, then *visible*$(u) := \emptyset$ and *hidden*$(u) := \emptyset$. In both cases the visible and hidden sets are updated for all nodes on the search path, and their corresponding edges are initiated or terminated if necessary. The interplay of endpoints and their vertical edges in the maintenance algorithms needs to be handled carefully.

An Efficient Representation

The visibility tree as described so far has two disadvantages. First, it requires disjoint-set union to be performed during updating, which because new copies are, apparently, required adversely affects the update time. Indeed a pessimistic estimate leads to an $O(n \log m)$ worst-case time bound for one update! Second, it apparently requires $O(n \log m)$ space. We overcome these disadvantages one at a time, resulting in Theorems 3.1 and 3.2 below.

To avoid copying of sets during a disjoint-union operation we keep only one global copy, essentially, of the hidden and visible sets associated with the root of the visibility tree. These are represented as doubly-linked lists, we denote them by *GH* and *GV*, respectively. Initially they are both empty. At each node u in the tree *hidden(u)* is represented by two pointers, *firsth(u)* and *lasth(u)*, which are both *nil* if *hidden(u)* is empty and, otherwise, point to the first and last elements of *hidden(u)* in *GH*. Thus the elements of *hidden(u)* must appear consecutively in *GH*. We will show this to be the case below. Similarly *visible(u)* is represented by two pointers *firstv(u)* and *lastv(u)*. However in this case they either both point into *GV* or both into *GH*. The reason for this is that *visible(u)* is the set of points 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

lastv(left(u)) should point to *firstv(right(u))* and vice versa

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 contenation. Thus the visible and hidden sets at nodes in $T(u)$ are unaffected by this operation.

Again we leave the details to the interested reader, only pointing out that we have replaced a putatively $O(n)$ time union with a constant time union. Moreover we have reduced the space requirements for *hidden(u)* and *visible(u)* to a constant rather than $O(n)$.

We overcome the second disadvantage by recognizing that we only need keep the size of *cover(u)* rather than *cover(u)* itself at each node u, cf. [LP]. Thus each node only requires constant space and deletion of an edge is now easily seen to be symmetric to insertion of an edge. In other words we no longer require the appearance linking of [BW].

Finally, 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 leaf requires constant time, and recomputing visible and hidden sets at an ancestor also requires constant time (the catenation of at most four lists), thus $O(\log m)$ time over all.

For actions (iii) and (iv) *COVERSEARCH* requires $O(\log m)$ time and the updating at all nodes visited during the search also requires $O(\log m)$ time. Finally the initiation or termination of e edges requires $O(e)$ time, giving $O(\log m+e)$ time overall. Thus we have proved:

Theorem 3.1 *A visibility tree with m leaves supports actions (i) and (ii) in* $O(\log m)$ *time, and actions (iii) and (iv) in* $O(\log m+e)$ *time, where e is the number of edges initiated or terminated.*

The space requirements for the visibility tree during *CONTOUR* consists of at most $O(n)$ for the cover sets, since we only need to keep the size of *cover(u)*. Moreover the visible and hidden sets also require $O(n)$ space, since each of the n points appears at most once in the global hidden and visible lists, and constant space is required at each node. Thus the tree requires $O(n)$ space since $2 \leq m \leq n$.

Combining this calculation with Theorem 3.1 we obtain:

Theorem 3.2 *Given a set of r-polygons consisting of n edges, the horizontal edges of the resulting disjoint r-polygons can be computed in* $O(n \log n+r)$ *time and* $O(n)$ *space, where r is the number of edges in the resulting r-polygons.*

Finally, if the given r-polygons are r-convex, then the space requirement can be reduced since we only need keep one spanning interval, rather than a sequence of adjacent vertical edges, for each r-polygon. Rather than inserting and deleting an edge directly, a new spanning interval is computed and inserted. Thus we can obtain:

Theorem 3.3 *Given a set of p r-convex r-polygons consisting of n edges, the horizontal edges of the resulting disjoint r-polygons can be computed in* $O(n \log n+r)$ *time and* $O(p)$ *space, where r is the number of edges in the resulting r-polygons.*

Acknowledgement:

I wish to thank Dr. Herbert Edelsbrunner for his valuable comments and his willing ear, which uncovered a fundamental error in an earlier version of this paper.

REFERENCES

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

[G1] Güting, R.H., An Optimal Contour Algorithm for Iso-Oriented Rectangles, *Journal of Algorithms* (1983), to appear.

[G2] Güting, R.H., Optimal Divide-and-Conquer to Compute Measure and Contour for a Set of Iso-Rectangles, Universität Dortmund, Lehrstuhl Informatik VI, Report 141, 1982.

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

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

[MF] Montuno, D.Y., and Fournier, A., Finding the $x-y$ Convex Hull of set of $x-y$ Polygons, University of Toronto, CSRG Technical Report 148, 1982.

[NLLW] Nicholl, T.M., Lee, D.T., Liao, Y.Z., and Wong, C.K., Constructing the $X-Y$ Convex Hull of a Set of $X-Y$ Polygons, *BIT* (1983), to appear.

[OWW] Ottmann, Th., Widmayer, P., and Wood, D., A Worst-Case Efficient Algorithm for Hidden Line Elimination, University of Waterloo Computer Science Technical Report CS-82-33, 1982.