Fast Algorithm for Polygon Decomposition

SURENDRA NAHAR AND SARTAJ SAHNI, FELLOW, IEEE

***Abstract*—We develop an $O(k \log (k) + n)$ algorithm, where n is the number of vertices in the polygon and k the number of vertical inversions, to decompose rectilinear polygons into rectangles. This algorithm uses horizontal cuts only and reports nonoverlapping rectangles whose union is the original rectilinear polygon. This algorithm has been programmed in Pascal on an Apollo DN320 workstation. Experimentation with rectilinear polygons from VLSI artwork indicates that our algorithm is significantly faster than the plane sweep algorithm and the algorithm proposed in [5].**

***Key words and phrases:* polygon decomposition, computational geometry, time complexity.**

I. Introduction

THE PROBLEM OF decomposing a polygon into basic components has applications in computer graphics, data bases, image processing, VLSI layout, and artwork analysis [5], [15], [13], [3]. The development of efficient algorithms to decompose a polygon has been the focus of much research. For example, Keil [8] develops polynomial time algorithms to decompose a simple polygon (i.e., one with no holes) into convex polygons, spiral polygons, star-shaped polygons, and monotone polygons. These algorithms minimize the number of simpler components without introducing any steiner points. Liu and Ntafos [10] consider the case when Steiner points are allowed. They develop a linear time algorithm to partition a simple monotone polygon into a minimum number of star-shaped polygons.

Asano *et al.* [1] present an $O(n^2)$ algorithm (n being the number of polygon vertices) to decompose a polygon with no holes into a minimum number of trapezoids. When the polygon has holes, this decomposition problem is NP-complete [4], [1]. In [11], an $O(n \log n)$ algorithm to partition a rectilinear polygon into a minimum number of uniformly monotone rectilinear polygons is developed. An $O(n^3)$ algorithm to find a maximum set of independent chords in a circle is used, in [12], to partition simple polygons into a minimum number of uniformly monotone polygons.

In this paper, we are concerned solely with the decomposition of rectilinear hole-free polygons into a minimum number of rectangles (cf. Fig. 1). Our work is trivially extendable to nonrectilinear polygons and also to polygons with holes.

The rectangles in the decomposition of a polygon are required to be disjoint (or nonoverlapping). When this restriction is removed (i.e., overlapping rectangles are allowed), decomposing a rectilinear polygon with holes into a minimum number of possibly overlapping rectangles is NP-complete. This is seen by observing that every 0, 1 matrix is the digitized version of some rectilinear polygon with holes (the 1's represent polygon interiors, the 0's exteriors). Hence the rectilinear picture compression problem [4] is the same as the polygon decomposition problem. The rectilinear picture compression problem is NP-complete.

Lingas *et al.* [9] use dynamic programming to obtain an $O(n^4)$ algorithm to dissect a rectilinear polygon with no holes into disjoint rectangles with minimum total edge length. They also show this to be NP-complete when holes are present. Ohtsuki [14] develops an $O(n^{5/2})$ algorithm to decompose a rectilinear polygon into a minimum number of rectangles using both horizontal and vertical cuts. This algorithm has been improved to $O(n^{3/2} \log n)$ in [7].

In some applications, only horizontal cuts are permissible. Fig. 2 shows the best decomposition when both horizontal and vertical cuts are permitted, as well as when only horizontal cuts are permitted. Consider any hole-free rectilinear polygon P. Let R_{HV} be the minimum number of rectangles in a decomposition using both horizontal and vertical cuts. Let R_H and R_V, respectively, denote the minimum number when only horizontal or only vertical cuts are permitted. It can be shown that

$$\min \{R_H, R_V\} < \tfrac{3}{2} * R_{HV}.$$

To see this, draw all possible vertical and horizontal line segments that join two internal corners (see Fig. 3). Let p_H be the number of horizontal line segments drawn, p_V the number of vertical segments, and p the number of lines in a maximum independent set of line segments (two line segments are independent iff they do not intersect). Clearly, $p \le p_V + p_H$. So, $p \le 2 * \max \{p_V, p_H\}$. From [14], it follows that

$$R_{HV} = H - p - 1$$

$$R_V = H - p_V - 1$$

$$R_H = H - p_H - 1$$

Manuscript received August 19, 1986; revised August 5, 1987, and November 6, 1987. This work was supported in part by the National Science Foundation under Grants DCR-8305567 and DCR-8420935. The review of this paper was arranged by Associate Editor R. H. J. M. Otten.

S. Nahar was with the Computer Science Department, University of Minnesota, Minneapolis. He is now with AT&T Bell Laboratories, Murray Hill, NJ.

S. Sahni is with the Computer Science Department, University of Minnesota, Minneapolis, MN 55455.

IEEE Log Number 8718828.

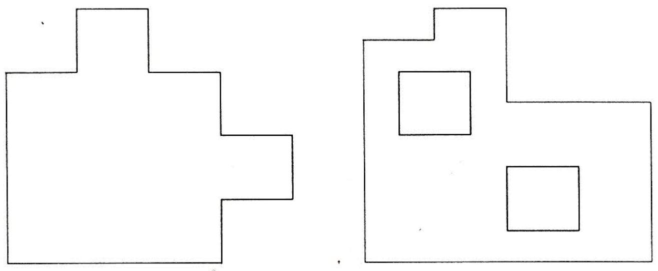

(a) (b)

Fig. 1. Rectilinear polygons. (a) No holes (hole-free). (b) With holes.

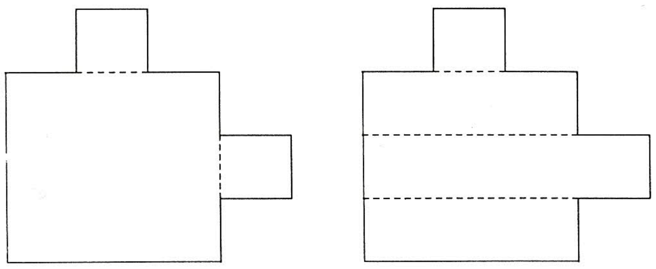

(a) (b)

Fig. 2. Rectilinear polygon decomposition. (a) Horizontal and vertical cuts. (b) Only horizontal cuts.

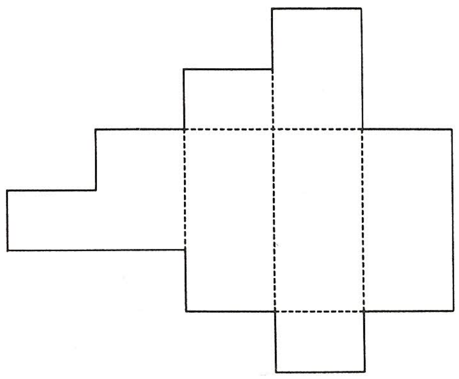


Fig. 3. All possible horizontal and vertical line segments that join two internal corners.

where H is the number of horizontal segments in the boundary of the original polygon. Hence,

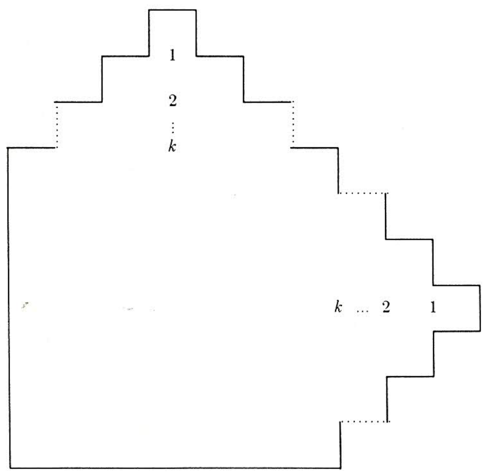
$$\min\{R_H, R_V\} = H - \max\{p_V, p_H\} - 1$$
$$\leq H - \frac{p}{2} - 1 \leq R_{HV} + \frac{p}{2}.$$

Also, since p segments create $p + 1$ polygons, $p < R_{HV}$. So,

$$\min\{R_H, R_V\} < \tfrac{3}{2} * R_{HV}.$$

Fig. 4 gives an example that approaches this bound asymptotically.

One application of horizontal cut decomposition is corner stitching [15]. This results in fast algorithms for interactive VLSI layout editing. An algorithm for horizontal



$H = 4 * k + 2,\ p = 2 * k$

$$\frac{\min\{R_H, R_V\}}{R_{HV}} = \frac{(4 * k + 2) - k - 1}{(4 * k + 2) - 2 * k - 1} = \frac{3}{2} - \frac{1}{(4 * k + 2)} = \frac{3}{2} \text{ as } k \to \infty$$

Fig. 4. An example where asymptotic bound is reached.

cut decomposition is given in [5]. This is reproduced in Fig. 5. This algorithm has a worst-case complexity of $O(n^2)$, though it is expected to do better than this on many polygons. An $O(n \log n)$ algorithm for horizontal cut decomposition is trivially obtained by performing a plane sweep using a vertical (or horizontal) scan line.

In this paper, we develop a new algorithm to decompose a hole-free rectilinear polygon into a minimum number of rectangles using only horizontal cuts. Our new algorithm takes advantage of the fact that polygons that arise in practice have a small number of vertical and horizontal inversions (defined below) relative to the number of vertices. The number of vertical inversions k_V of a hole-free rectilinear polygon is obtained by traversing the vertices of the polygon in anticlockwise order. This traversal begins at a vertex with least y coordinate. Of the several vertices that have this y value, the one with least x is chosen. This vertex, denoted σ, is called the start or *σ vertex.*

Consider the sequence of distinct y values encountered in the traversal. This sequence is initially increasing, then decreasing, then increasing, then decreasing, etc. The tail end of this sequence is always decreasing. Each time the sequence changes from increasing to decreasing, there is a *vertical inversion.*

Example: For the polygon of Fig. 6(a), the y value sequence is $y_1 < y_2 < y_3 < \cdots < y_{14} > y_{15} \cdots > y_{20}$. There is one vertical inversion in the sequence. Fig. 6(b) shows a polygon with two vertical inversions and Fig. 6(c) shows one with three vertical inversions. □

The number of horizontal inversions is defined in an analogous manner. This time, the anticlockwise traversal begins at a vertex with least x value. Of the several vertices that have this x value, the one with the lowest y value is used. This start vertex is called the *τ vertex.* We consider the sequence of distinct x values encountered during the traversal. This sequence is initially increasing, then decreasing, then increasing, then decreasing, etc. The tail

```
Step 1   Let the given polygon be represented by a set V of vertices
           $(x_i, y_i)$
Step 2   Let $P_k$ be the leftmost of the lowest vertices in V; $P_l$ the next
           leftmost of the lowest vertices in V
Step 3   Let $P_m$ be the leftmost of the lowest vertices in V with y value
           greater than $y_k$ and x coordinate in the range between $x_k$ and
           $x_l$ (including $x_k$ and $x_l$)*
Step 4   The next rectangle is: $x_{\min} = x_k$; $y_{\min} = y_k$; $x_{\max} = x_l$; $y_{\max} =$
           $y_m$
Step 5   Remove $P_k$ and $P_l$ from V. Remove $(x_k, y_m)$ and $(x_l, y_m)$ if
           present in V else add them to V
Step 6   If V is empty then stop else goto step 2
         *Gourley and Green exclude $x_l$. $x_l$ has to be included to work
           for all cases of this problem.
```

Fig. 5. Algorithm of Gourley and Green [5] to dissect hole-free rectilinear polygons using horizontal lines.

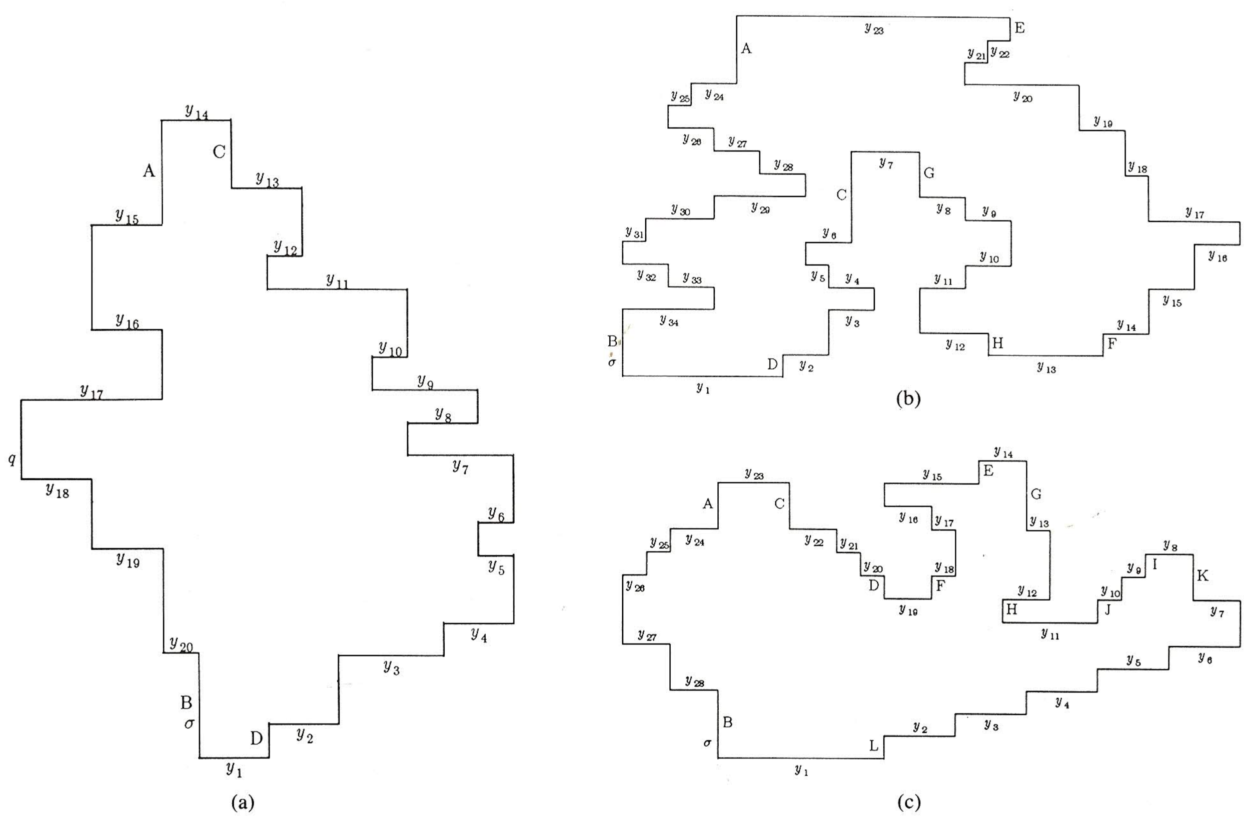


Fig. 6. Polygons having different numbers of inversions. Start vertex for vertical inversions is labeled σ. Start vertex for horizontal inversions is labeled q. (a) $k_V = 1$, $k_H = 6$. (b) $k_V = 2$, $k_H = 6$. (c) $k_V = 3$, $k_H = 3$.

end of the sequence is always decreasing. Each time the sequence changes from increasing to decreasing, there is a *horizontal inversion*. Let k_H be the number of horizontal inversions. For the examples of Fig. 6(a), (b), and (c), $k_H = 6$, 6, and 3, respectively. We define the number of inversions k to be: $k = \min\{k_V, k_H\}$. For the examples of Fig. 6(a), (b), and (c), $k = 1$, 2, and 3, respectively.

Table I gives inversion statistics for a sample of 2869 polygons taken from VLSI mask data provided by Sperry Corp. Table II gives the distribution of polygons in this sample set by the number of vertices. As can be seen, the number of inversions k is "small." In fact, over 85 percent of the polygons have only one inversion (i.e., $k = 1$) and over 95 percent have at most two (i.e., $k \leq 2$)! Neither the scan line method nor that of [5] attempts to take advantage of this observation. In the next section we describe a new algorithm for horizontal cut decomposition that does take advantage of the observation that many rectilinear polygons that arise in practice have a small number (k) of inversions. The complexity of this algorithm is $O(k \log(k) + n)$. Experimental data provided in Section III indicate that our algorithm is superior to the scan line method and to the algorithm of [5] when k is small relative to n.

TABLE I
INVERSION STATISTICS FOR A SAMPLE OF 2869 POLYGONS TAKEN FROM VLSI MASK DATA PROVIDED BY SPERRY CORPORATION

#of inversions i	#of polygons $k_V = i$	#of polygons $k_H = i$	#of polygons $k = i$
1	2315	1536	2474
2	385	761	290
3	93	373	61
4	40	71	28
5	4	50	4
6	0	51	12
7	0	27	0
8	0	0	0
9	0	0	0
10	16	0	0
11	0	0	0
12	0	0	0
13	0	0	0
14	16	0	0

TABLE II
DISTRIBUTION OF POLYGONS BY NUMBER OF VERTICES

Number of vertices	Number of polygons
$0 < n \leq 10$	1739
$10 < n \leq 20$	926
$20 < n \leq 30$	84
$30 < n \leq 40$	25
$40 < n \leq 90$	91

II. OUR ALGORITHM

A. Terminology

Definition: A *vertical wall* is a maximal boundary of a polygon that satisfies the following:

1) The first and last edges of the vertical wall are vertical edges. The first edge is called the *top* of the wall and the last edge the *bottom*.
2) The y coordinates of the horizontal edges, if any, of the vertical wall decrease from top to bottom.

The polygon of Fig. 6(a) has two vertical walls. The top of the first vertical wall is labeled A and its bottom B in the figure. The top of the second vertical wall is labeled C and its bottom is labeled D. The polygon of Fig. 6(b) has four vertical walls. Their tops are labeled A, C, E, and G while their bottoms are labeled B, D, F, and H, respectively. The polygon of Fig. 6(c) has six vertical walls. The tops are A, C, E, G, I, and K and the bottoms B, D, F, H, J, and L. It is easy to prove the following lemmas.

Lemma 1: A rectilinear polygon has k vertical inversions iff it has $2 * k$ vertical walls. □

Lemma 2: No two vertical walls of a rectilinear polygon intersect. □

Definition: A *left* vertical wall is one for which the area immediately to the right of its vertical edges is part of the polygon interior (or, equivalently, the area immediately to its left is exterior to the polygon). A vertical wall that is not a left vertical wall is a *right* vertical wall. □

A vertical wall with top edge A and bottom edge B is called an AB vertical wall. Wall AB of Fig. 6(a) is a left vertical wall while vertical wall CD is a right vertical wall. The left vertical walls of Fig. 6(b) are AB and GH and the right vertical walls are CD and EF. The vertical walls AB, EF, and IJ of Fig. 6(c) are left vertical walls while walls CD, GH, and KL are right vertical walls.

Lemma 3: If the boundary of a rectilinear polygon is traversed in anticlockwise order, starting at the σ vertex, then the first vertical wall encountered is a right vertical wall, the next a left, the next a right, and so on. Left and right vertical walls alternate in this traversal. □

Let XY be a vertical wall. X is its top edge and Y its bottom edge. $y$top(XY) is the maximum y coordinate of X (note X is a vertical edge). $y$bottom(XY) is the minimum y coordinate of Y. $x$top(XY) and $x$bottom(XY) are, respectively, the x coordinate of X and Y. x_coord(XY, y) is such that (x-coord(XY, y), y) is a point on a vertical edge of XY. If there are two possibilities for the value of x_coord(XY, y), then either may be used.

Definition: Let XY and WZ be two vertical walls. XY is to the left of WZ iff either

$$y\text{bottom(XY)} \leq y\text{top(WZ)} \leq y\text{top(XY)}$$
and
$$x\_\text{coord(XY}, y\text{top(WZ))} < x\text{top(WZ)}$$

or

$$y\text{bottom(WZ)} \leq y\text{top(XY)} \leq y\text{top(WZ)}$$
and
$$x\text{top(XY)} < x\_\text{coord(WZ}, y\text{top(XY))}. \quad \square$$

Fig. 7 shows two examples of vertical wall pairs XY and WZ such that XY is to the left of WZ. We write XY $<$ WZ when XY is to the left of WZ. Fig. 8 shows two examples of vertical wall pairs XY and WZ such that neither XY $<$ WZ nor WZ $<$ XY.

Let $<^+$ be the transitive closure of $<$ (i.e., XY $<^+$ WZ iff there exists a (possibly empty) sequence of vertical walls such that XY $< \cdots <$ WZ). It is easy to see that $<^+$ defines a partial order on the vertical walls of any rectilinear polygon [6] (to see this, we merely observe that $<^+$ is irreflexive and transitive).

Horizontal walls and their associated terminology are defined analogously.

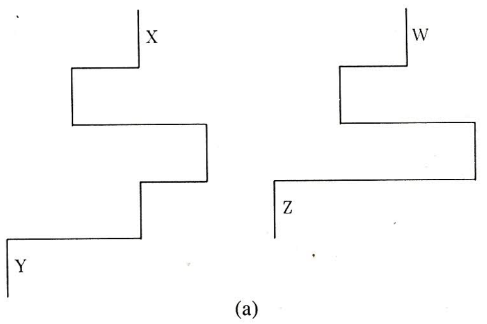
Definition: A *horizontal wall* is a maximal boundary of a polygon that satisfies the following:

(i) The first and last edges of the wall are horizontal edges. The first edge is called the *left end* of the wall and the last the *right end*.
(ii) The x coordinates of the vertical edges (if any) increase from left to right.

An *upper wall* is a horizontal wall for which the area immediately below the horizontal edges is part of the polygon interior. A horizontal wall that is not an upper wall is a *lower wall*. □

Lemmas 1, 2, and 3 have their obvious analogues for horizontal walls.

Because the overwhelming majority of polygons used



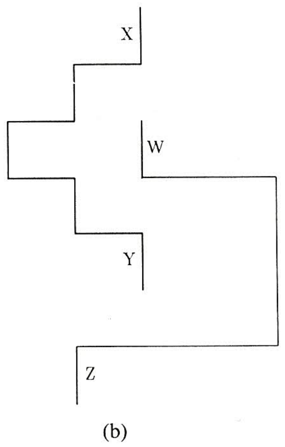


Fig. 7. Examples of vertical wall pairs XY and WZ such that XY is to the left of WZ, i.e., XY < WZ.

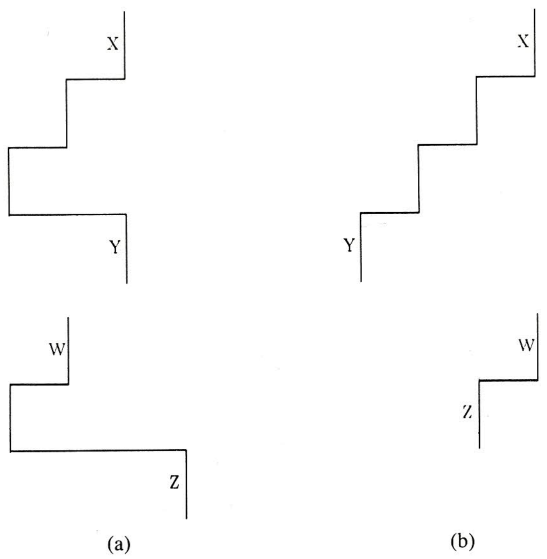


Fig. 8. Examples of vertical wall pairs which do not interact. Neither XY < WZ nor WZ < XY.

in practice are expected to have a $k = \min\{k_V, k_H\}$ of 1 of 2, we treat these as special cases. At the top level, our algorithm is divided into two parts: one for the case $k = k_V$ and the second for the case $k = k_H$ (in case $k_V = k_H$, either part may be used). For each of these parts, there are three subparts: one for the case $k = 1$, another one for the case $k = 2$, and the third for the case $k > 2$.

B. $k = k_V$

1) $k = 1$: When $k = 1$, the polygon has two vertical walls. One is a left vertical wall and the other is a right vertical wall. The rectangular decomposition may be obtained by starting at the bottom of each vertical wall and advancing to the top. During this vertical wall traversal, we use two pointers i and j. The pointer i traverses the vertical edges of the left vertical wall and j those of the right vertical wall.

Consider the partial polygon of Fig. 9. Initially, $i = a$ and $j = 1$. The first rectangle is obtained by drawing a horizontal edge from the top of 1 to edge a. This is a correct rectangle as there can be no vertical wall between the left and right vertical walls being traversed. The pointer j is advanced to the next vertical edge, 2, on the right vertical wall. At this time, a horizontal edge to 2 is drawn from the top of a and i advanced to b. Since the tops of i and j edges are the same, i and j are both advanced following the drawing of the next rectangle. This process continues until the tops of the left and right vertical walls are reached.

2) $k = 2$: Polygons with $k = 2$ have four vertical walls (Lemma 1). When the boundary of the polygon is traversed in anticlockwise order beginning at the σ vertex, the vertical walls are encountered in the order R1, L1, R2, L2, where R1 and R2 are right vertical walls and L1 and L2 are left vertical walls (Lemma 3). At the top of vertical wall R1, there are two choices for the direction of the horizontal edge (left and right). If the polygon moves right at the top of R1, then at the bottom of L1 it must also move right. Otherwise, the remaining two vertical walls cannot close the polygon. If the polygon moves left at the top of R1, then at the bottom of L1, it may move either left or right. Once this choice has been made, the direction at the top of R2 is determined. The three cases are shown in Fig. 10.

The three cases of Fig. 10 are handled separately as described below:

Case (a)

(i) Start at the bottom of L2 and R1 cutting off rectangles as for the case $k = 1$. Stop when the top of R1 is reached.
(ii) Repeat (i) using L1 and R2. Stop when the top of L1 is reached.
(iii) Process the remainder of L2 and R2.

Case (b)

(i) Start at the bottom of L2 and R1 cutting off rectangles as for the case $k = 1$. Stop when the bottom of R2 (or L1) is reached.
(ii) Process the remainder of L2 and R2 as for the case $k = 1$.
(iii) Process the remainder of R1 and L1 as for the case $k = 1$.

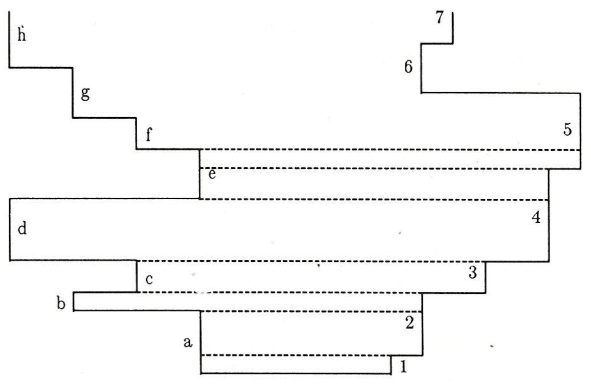


Fig. 9. Portion of a $k = 1$ polygon.

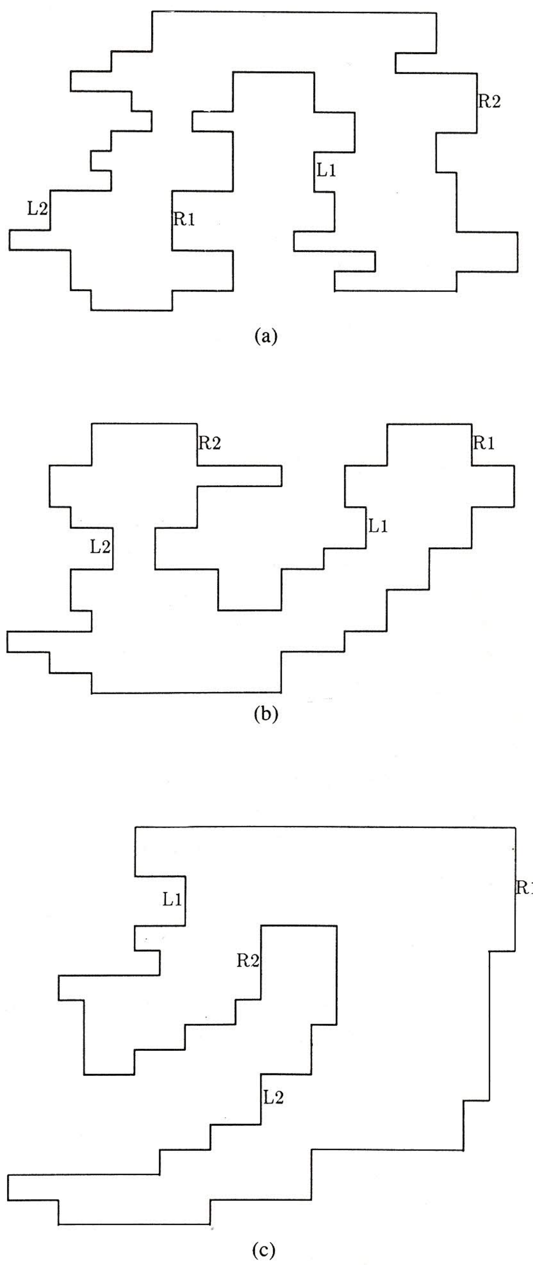


Fig. 10. Polygon decomposition: cases for $k = 2$. (a) Move right at top of wall R1. (b) Move left at the top of wall R1 and move left at the bottom of wall L1. (c) Move left at the top of wall R1 and move right at the bottom of wall L1.

Case (c)

(i) Start at the bottom of L2 and R1 cutting off rectangles as for the case $k = 1$. Stop when the top of L2 is reached.
(ii) Process L1 and R2 starting at the bottom. Stop when the top of R2 is reached.
(iii) Process the remainder of L1 and R1.

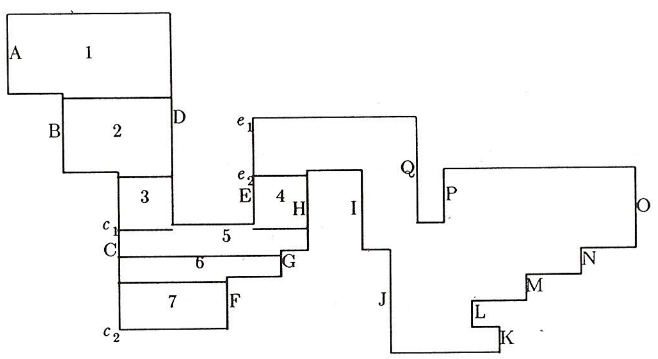
3) $k > 2$: While we could develop custom algorithms for each of the cases $k = 3$, $k = 4$, $\cdots$, $k = q$, the statistics of Table I indicate that not much is to be gained beyond $k = 2$. Hence, for $k > 2$ we develop a general algorithm. The major steps in this general algorithm are described below:

Step 1: Traverse the polygon in counterclockwise order determining the top and bottom of each vertical wall. This traversal may begin at any vertex. Following this traversal, the σ vertex may be identified.
Step 2: If the number k of vertical inversions is 1 or 2 (i.e., number of vertical walls is less than 6), then the decomposition is obtained using the strategy of Sections II-B-1 and II-B-2. So, assume $k > 2$.
Step 3: For each pair of vertical walls i, j determine if $i < j$ or $j < i$.
Step 4: Obtain a topological order from the partial order specified by the '$<$' relation constructed in step 3.
Step 5: Obtain the rectangular decomposition by processing the vertical walls in the topological order of step 4.

An example will help illustrate the details of the processing that is involved. Consider the polygon of Fig. 11(a). It has the eight vertical walls AC, HF, IJ, OK, PP, QQ, EE, and DD. The relation $<$ is shown in Figure 11(b). A topological order for the vertical walls is: AC, DD, EE, HF, IJ, QQ, PP, OK. It should be evident that the first vertical wall in topological order will always be a left vertical wall. This first vertical wall AC is the *current* left vertical wall.

The next vertical wall in the topological order is DD. Since DD is a right vertical wall, we can process the current vertical wall and DD to cut off the rectangles 1, 2, and 3 shown in Fig. 11(a). Following this, the current left vertical wall consists of just the segment c_1, c_2.

The next vertical wall in the topological order is EE. Since EE is a left vertical wall, it is just appended to the current left vertical wall. Actually, the current left vertical wall is maintained as a collection of vertical wall segments, in this case c_1, c_2 and EE. Next, the vertical wall HF is encountered. As it is a right vertical wall, it may be processed with respect to the current left vertical wall to get the rectangles 4, 5, 6, and 7 of Figure 11(a). Following this, the current left vertical wall consists of the segment e_1, e_2. Next, IJ is appended to the current left vertical wall. Then QQ is processed and so on.



(a)

R	AC	DD	EE	HF	IJ	QQ	PP	OK
AC	*	<	<	<	<	<	<	<
DD	*	*	<	<	<	<	<	<
EE	*	*	*	<	<	<	<	<
HF	*	*	*	*	<	<	<	<
IJ	*	*	*	*	*	<	<	<
QQ	*	*	*	*	*	*	<	<
PP	*	*	*	*	*	*	*	<
OK	*	*	*	*	*	*	*	*

(b)

Fig. 11. (a) Example polygon decomposition to illustrate major steps in general algorithm ($k > 2$). (b) The relation $<$ for the walls of (a).

The correctness of the procedure described above follows from the fact that when the current left vertical wall and a right vertical wall are processed, there are no vertical walls in between. This is guaranteed by the topological order obtained from the relation '$<$'.

4) Implementation and Complexity: We assume that the input is an array of polygon edges in anticlockwise order. From this, k can be determined in $O(n)$ time by traversing the edges in the input order. During this traversal, the top (and hence bottom) of each vertical wall can be identified. With this information the case $k = 1$ is easily solved in $O(n)$ time. If $k = 2$, we can resolve which one of the cases (a), (b), or (c) the given polygon falls into by determining the direction the polygon takes at the top of R1 and bottom of L1. The processing of each case can easily be done in $O(n)$ time. For the case $k > 2$, we describe two different implementations. The first uses simple data structures and is suitable when k is small. The second uses more complex data structures, has a smaller asymptotic complexity, and is suitable when k is large.

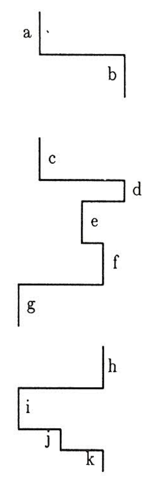
a) Implementation for small k: When k is small, the relation $i < j$ may be constructed in $O(k^2 \log (n/k))$ time. There are $k(k-1)/2$ pairs (i, j) to be considered and the computation of x_coord can be done in $O(\log m)$ time using a binary search on a vertical wall with m vertical segments. The topological ordering of the vertical walls can be done in $O(k^2)$ time (see [6] for example). This leaves us with the processing of step 5.

To implement step 5 efficiently, we need a good data structure for the current left vertical wall. The current left vertical wall consists of segments of original left vertical walls. These segments are kept track of in a table CurrentLeftWall. Each entry of this table has three fields: first, second, and last, where CurrentLeftWall[i].X points to the Xth edge of the ith segment of the current vertical wall, $X \in \{\text{first, second, last}\}$. This table is maintained so that segments are in decreasing order of y. Fig. 12 gives an example current left vertical wall and the CurrentLeftWall table. There are two cases to consider for left vertical wall processing. The first is when the next vertical wall in topological order is a right wall. At this time the matching left segments need to be found. The topmost such segment may be found in $\log k$ time using a binary search on CurrentLeftWall. Next the matching edge within this segment is to be found. This takes $O(\log m)$ time if a binary search is used within the segment (m is the number of vertical edges in the segment). Following the processing, a vertical wall segment may be split into two (see Fig. 13). The field *first* enables us to handle the case of Fig. 13(b) without copying the new segment into a new part of memory. The actual cutting of rectangles is linear in the number of rectangles cut.

When the next vertical wall is a left vertical wall, we need to insert an entry into the CurrentLeftWall table. This takes $O(k)$ time as there can never be more than $2 * k$ vertical wall segments. Hence the time to process the next vertical wall is $O(k + \log m + \#$ of rectangles cut$)$. So, the total time for step 5 is $O(k^2 + k \log n + n)$. Combining with the time for the other steps, we get $O(k^2 \log (n/k) + n)$ as the complexity of the new algorithm when the small k implementation is used.

b) Implementation for large k[1]*:* When k is large, it is advantageous to construct the partial order '$<$' by using a horizontal scan line that begins at the top of the polygon. All vertical walls cut by this scan line are maintained in a balanced binary search tree (e.g., AVL or 2-3 tree, [6]). An in-order traversal of this search tree visits the walls in the order (left to right) they are cut by the scan line. Observe that since vertical walls do not cross one another, this relative order is unchanged as the scan line moves. An *event* occurs when the scan line reaches the top or bottom of a wall. If both the tops of some walls and the bottoms of others are reached at the same y value, then the tops are processed before the bottoms. To process the tops, we note that the walls whose tops are reached are currently not in the binary search tree. These walls need to be inserted into the search tree so that the in-order sequence corresponds to the left to right order in which the scan line cuts these walls. This is accomplished by using the standard insertion algorithm for balanced binary search trees. To compare the x coordinates of two walls, we need to walk down a wall till the appropriate y segment is reached. During this walk, wall segments above the current scan line are discarded. In addition to the insertion, wall pairs $W_a < W_b$ are to be generated. Let $W_1, W_2, W_3, \cdots$ be the in-order sequence after the in-

[1]This implementation was suggested by an anonymous referee.

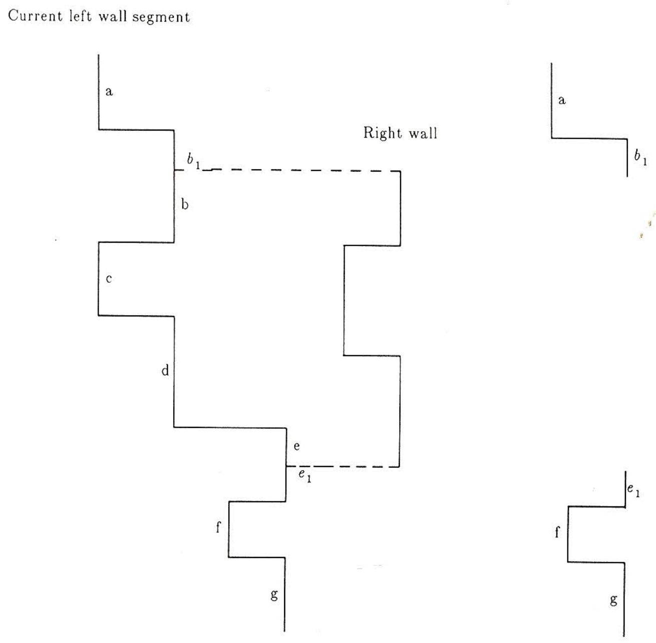


(a)

first	second	last
a	b	b
c	d	g
h	i	k

(b)

Fig. 12. An example of current left vertical wall and its CurrentLeftWall table. (a) Current left wall segments. (b) Table.



(a) (b)

Fig. 13. A vertical wall may split into two during processing. (a) Current left wall segments. (b) After processing.

sertion of the new walls. The pairs to be generated are $W_i < W_{i+1}$ for all i such that

W_i is an old wall and W_{i+1} is a new wall

or

W_i and W_{i+1} are both new walls

or

W_i is a new wall and W_{i+1} is an old wall.

Processing wall bottoms simply requires the deletion of the walls from the tree. Since individual insertions and deletions into a balanced binary search tree take logarithmic time and since the total number of wall segments we need to walk down is $O(n)$, all the '$<$' pairs can be generated in $O(k \log k + n)$ time. The topological order can be constructed in less time than this [6].

The implementation of step 5 is essentially the same as that for the case 'k small' except that a balanced tree is used for the current wall rather than a table. With this change, step 5 also runs in $O(k \log k + n)$ time.

C. $k = k_H$

1) $k = 1$: At this time, the polygon has two horizontal walls. We begin at the left end of each wall. The vertical edge that joins these two left ends is the *CurrentLeftWall.* In general, the current left wall will consist of several vertical edges (or portions of vertical edges). These are maintained as a sequentially ordered list (ordered top to bottom). Two pointers i and j are used to advance along the upper and lower walls. Since $k_H = 1$, there are no horizontal walls between these two and the rectangular decomposition can be done in a straightforward way.

Consider the example of Fig. 14. Initially, the CurrentLeftWall is (1, 3), $i = a$ and $j = l$. Since the right end of j is less than that of i, j is advanced to m. The vertical edge between l and m is a right edge (i.e., the interior is to its left) and so a rectangle is cut off by drawing a broken line from the left end of m to the CurrentLeftWall. This wall is now updated to be (1, 2). Next, i is advanced to b (as the right end of $i = a$ is $\leq$ the right end of $j = m$) and the CurrentLeftWall becomes (5, 6), (1, 2). Now, j is advanced to n and the CurrentLeftWall updated to (5, 6), (1, 2), (7, 8). j is again advanced; this time to o. Since the vertical edge between n and o is a right edge, rectangles can be cut off. This is done by moving up the CurrentLeftWall. First, a rectangle is cut off using the (7, 8) segment. Then one is cut off using the (1, 2) segment and finally a third one is cut off using part of the (5, 6) segment. The CurrentLeftWall becomes (5, 4) and j is advanced to p. Processing terminates when i and j reach the right end of their respective walls.

The processing described above is quite similar to that for the case $k_V = 1$. The major difference is the addition of a list for the CurrentLeftWall. As remarked earlier this may be maintained as a sequential list as CurrentLeftWall is essentially a deque[2] [6]. The scheme described may be implemented so as to have time complexity $O(n)$. However the constant factor associated is slightly higher than for the case $k_V = 1$. So, if a polygon has $k = k_V = k_H = 1$ then it should be handled using the $k_V = 1$ algorithm.

2) $k = 2$: Now there are four horizontal walls. Two upper U1 and U2 and two lower L1 and L2. We may label these by traversing the polygon beginning at the τ vertex. As in the case for $k_V = 2$, there are three cases to consider. In the first, the polygon moves down at the right end of the first lower wall L1. Following this, the polygon must move left and then down at the end of upper U1. At the end of this, it must move right. From the right end of

[2]A deque is a linear list which permits additions and deletions from either end.

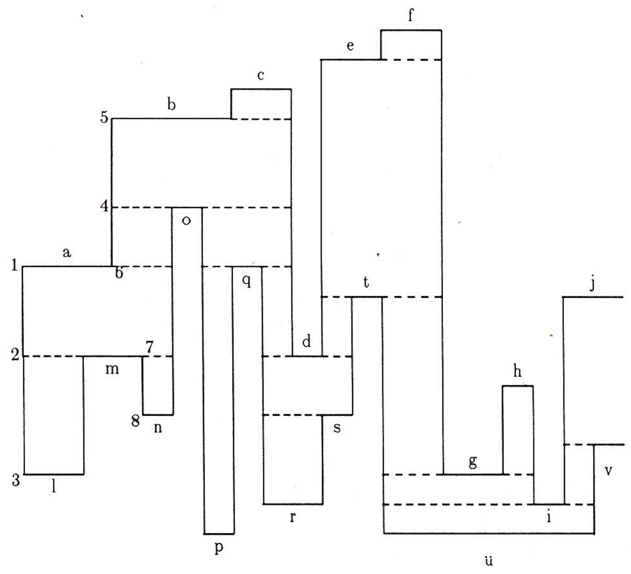


Fig. 14. Polygon decomposition example for $k_H = 1$.

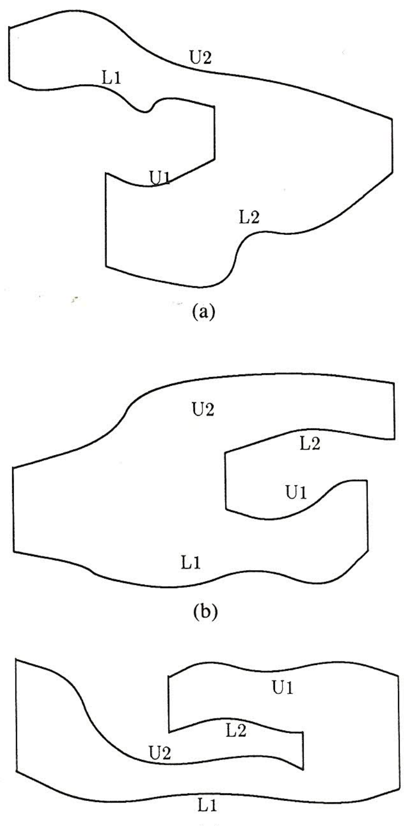


Fig. 15. Polygon decomposition: cases for $k = 2$.

the second lower wall L2, U2 must move left. This case is shown in Fig. 15(a). When the polygon moves up at the end of L1, there are two cases depending on whether the polygon moves up or down at the left end of U1. These two cases are shown in Fig. 15(b) and (c).

The processing of these three cases is similar to that for the case $k_H = 1$. We describe this only for case (a).

Case (a)

(i) Process L1 and U2 left to right as for $k_H = 1$. Stop when the right end of L1 is reached.

(ii) Process U1 and L2 as for $k_H = 1$ and stop when the right end of U1 is reached.

(iii) Process the remainder of U2 and L2. To do this, the CurrentLeftWalls following (i) and (ii) need to be appended to each other.

Once again, it is easy to see that the required processing can be done in $O(n)$ time.

3) $k > 2$: When $k > 2$, we order the horizontal walls by their left end points. This is done by sorting these points into nondecreasing order of x and within x into nondecreasing order of y. The walls now form an alternating sequence of lower and upper walls. We begin with all pairs of matching lower and upper walls with the leftmost left end point. These are the first few pairs in the sorted order just obtained. The left ends of the upper and lower walls in each such pair are connected by a vertical edge in the polygon. This edge defines the CurrentLeftWall for that pair.

The pairs of upper and lower walls are processed as for the case $k_H = 1$. The processing continues until the leftmost x coordinate of the next wall pair in the sorted order is reached or until a wall is exhausted. If the former happens, then the new wall pair either causes a CurrentLeftWall to be split into two or a new CurrentLeftWall to be added to the collection of active left walls. If the latter happens, then either two CurrentLeftWalls get combined or one becomes empty.

Consider the example polygon of Fig. 16. Following the sort, the wall ordering is W_4, W_3, W_2, W_1, W_8, W_7, W_6, W_5, W_9, W_{10}, $\cdots$.

We begin with the two matched pairs (W4, W3) and (W2, W1). The CurrentLeftWall for (W4, W3) is $\{(5, 6)\}$ and that for (W2, W1) is $\{(1, 2)\}$. These wall pairs are processed independently. During this processing a rectangle with left edge (6, 7) is cut off first and CurrentLeftWall for (W4, W3) updated to $\{(5, 7)\}$. Next, the right ends of W2 and W3 are reached and the two left walls merged to get $\{(1, 2), (3, 4), (5, 7)\}$. Next the pair (W7, W8) is encountered. This requires us to split the CurrentLeftWall by cutting off rectangle 8, 9, 10, 11. We now process the pairs (W7, W1) and (W4, W8) independently. The CurrentLeftWalls are, respectively, $\{(12, 13), (1, 2), (3, 10)\}$, and $\{(11, 4), (5, 7)\}$. When the pair (W9, W10) is reached, a new matched pair is added to the collection of wall pairs being processed.

By using appropriate data structures, the scheme just described can be implemented to have a time complexity that is $O(k \log n + n)$. This implementation requires the use of balanced search trees to allow fast searching in a CurrentLeftWall. Use of an AVL tree [6], for example, permits efficient search, splitting, and combining. In practice, we do not expect the use of a balanced search tree to provide good performance as we expect k to be

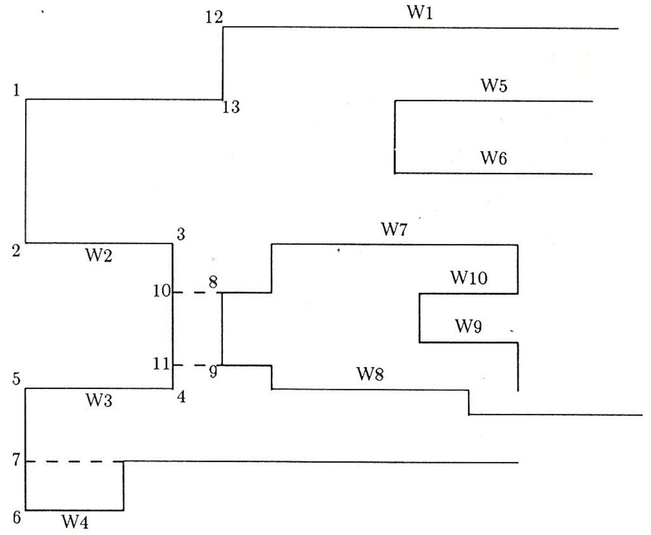


Fig. 16. Example polygon decomposition for $k > 2$.

'small,' In this case, a more direct implementation with complexity $O(k^2 + n)$ is expected to perform better.

The overall complexity of our algorithm is seen to be $O(\min\{k_V \log(k_V) + n, k_H \log n + n\})$. Since k_V and k_H can be as large as $O(n)$, this becomes $O(n \log n)$. This is comparable to the plane sweep method and is superior to the method of Gourley and Green [5]. However for every fixed k, we expect our algorithm to outperform the others when n is suitably large.

While our description of the algorithm requires the use of the codes of Sections II-B-1, II-B-2, and II-B-3 whenever $k_V \leq k_H$, in practice we may do better using some other decision rule. Depending on the particular implementation of the algorithms described, it is quite possible that for polygons with $k_V = 6$ and $k_H = 8$ (for example), the code for the algorithm of Section II-C-3 is faster.

III. Experimental Results

We have programmed all three of the rectangle decomposition algorithms in Pascal. For vertical inversions, we considered only the case of small k. Even for this, our implementation differs from the small k implementation described in Section II-B-4 in that the CurrentLeftWall is maintained as a two entry table (first and last) rather than a three entry table (cf. Fig. 12(b)). This implementation results in an increased asymptotic complexity. However, since k_V is small for our test set the number of segments in the CurrentLeftWall is also small and the coded version of the algorithm faster than the asymptotically superior algorithm described in Section II-B-4.

Run times have been obtained on an Apollo DN320 workstation using the 2869 polygon sample set described in Tables I and II. Table III gives the observed run times. All times are in milliseconds (ms) and are the sum of the times for all polygons in the given category. For example, the plane sweep algorithm took 1767 ms to decompose the 1739 polygons (cf. Table II) with up to 10 vertices. The column labeled "Vertical inversion k_V" gives the time taken when the vertical inversion algorithm of Section II-B is used irrespective of whether k_V or k_H is smaller; the column labeled "Horizontal inversion k_H" gives the time taken when the horizontal inversion algorithm of Section II-C is used irrespective of whether k_V or k_H is smaller; the column labeled "min $\{k_V, k_H\}$" gives the time taken when the vertical inversion algorithm is used if $k_V \leq k_H$ and the horizontal inversion algorithm used otherwise; the last column gives the time required if the faster of the two inversion algorithms is used on each instance (i.e., each instance is solved by the inversion algorithm that will solve it faster). As can be seen, there is not much difference in the last two columns. So, for our implementation choosing between the two inversion algorithms as described in Section II is near optimal. The plane sweep algorithm is significantly faster than that of [5]. The inversion algorithm of Section II is, in turn, significantly faster than the plane sweep algorithm. In fact, the inversion algorithm (see column labeled min $\{k_V, k_H\}$) required less than half the time required by the plane sweep algorithm to decompose the 2869 polygon test set.

TABLE III
Performance Comparison of Various Algorithms Using VLSI Mask Data Provided by Sperry Corporation

Number of vertices (n)	Time taken by algorithms in milliseconds					
	[GOUR83]	Plane sweep	Vertical inversion k_V	Horizontal inversion k_H	min $\{k_V, k_H\}$	min time $\{k_V, k_H\}$
$0 < n \leq 10$	4349	2803	1181	1767	1214	1162
$10 < n \leq 20$	4794	2662	1311	3325	1051	1049
$20 < n \leq 30$	1146	522	396	657	289	289
$30 < n \leq 40$	693	274	77	375	77	77
$40 < n \leq 90$	4969	1440	2350	1458	1115	1114
Total	15953	7703	5316	7584	3747	3692

IV. Conclusions

We have developed a fast algorithm for the decomposition of hole-free rectilinear polygons. Experimental results obtained using VLSI mask data indicate our algorithm is superior to previously known algorithms for this problem. The method used by our algorithm is trivially applicable to the case of nonrectilinear polygons (trapezoidal decomposition) and polygons with holes.

References

[1] T. Asano, T. Asano, and H. Imai, "Partitioning a polygonal region into trapezoids," *J. Ass. Comput. Mach.*, vol. 33, no. 2, pp. 290–312, Apr. 1986.

[2] S. Chaiken, D. J. Kleitman, M. Saks, and J. Shearer, "Covering regions by rectangles," *SIAM J. Algebraic and Discrete Methods*, vol. 23, no. 4, pp. 394–410, Dec. 1981.

[3] J. P. Cohoon, "Fast channel graph construction," Department of Computer Science, University of Virginia, 1985.

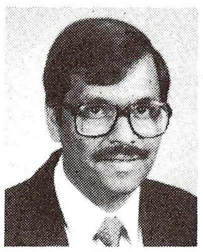
[4] M. R. Garey and D. S. Johnson, *Computers and Intractability.* San Francisco: W. H. Freeman, 1979.

[5] K. D. Gourley and D. M. Green, "A polygon-to-rectangle conversion algorithm," *IEEE Computer Graphics*, vol. 3, no. 1, pp. 31–36, Jan./Feb.

[6] E. Horowitz and S. Sahni, *Fundamentals of Data Structures In Pascal.* Rockville, MD: Computer Sci. Press, 1984.

[7] H. Imai and T. Asano, "Efficient algorithms for geometric graph search problems," *SIAM J. Comput.*, vol. 15, no. 2, pp. 478–494, May 1986.

[8] J. M. Keil, ''Decomposing a polygon into simpler components,'' *SIAM J. Comput.*, vol. 14, no. 4, pp. 799-817, Nov. 1985.
[9] A. Lingas, R. Y. Pinter, R. L. Rivest, and A. Shamir, ''Minimum edge length decomposition of rectilinear polygons,'' Extended Abstract, Massachusetts Institute of Technology, 1981.
[10] R. Liu and S. Ntafos, ''On partitioning rectilinear polygons into star-shaped polygons,'' University of Texas at Dallas, UTD Tech. Rep. #216, 1985.
[11] R. Liu and S. Ntafos, ''Partitioning rectilinear polygons into rectilinear parts,'' University of Texas at Dallas, UTD Tech. Rep. #217, 1985.
[12] R. Liu and S. Ntafos, ''On decomposing polygons into uniformly monotone parts,'' University of Texas at Dallas, UTD Tech. Rep. #218, 1985.
[13] S. Nahar and S. Sahni, ''A time and space efficient net extractor,'' in *Proc. 23rd Design Automat. Conf.*, 1986, pp. 411-417.
[14] T. Ohtsuki, ''Minimum dissection of rectilinear regions,'' in *Proc. 1982 Int. Symp. Circuits Syst.*, 1982, pp. 1210-1213.
[15] J. K. Ousterhout, ''Corner stitching: A data-structuring technique for VLSI layout tools,'' *IEEE Trans. Computer-Aided Design*, vol. CAD-3, no. 1, pp. 87-100, Jan. 1984.



Surendra Nahar received the Bachelor of Technology degree in electrical engineering from the Indian Institute of Technology, Kanpur, in June 1982. He received the M.S. and Ph.D. degrees in computer science from the University of Minnesota in March 1985 and June 1986, respectively. From June 1983 to June 1986, he worked part-time in the Computer Aided Design Department at Sperry Corporation (Unisys), Minneapolis, MN, developing CAD algorithms.

He joined AT&T Bell Laboratories, Murray Hill, NJ, in July, 1986, where he is a Member of the Technical Staff in the Computer Aided Design and Test Laboratory. He is currently working on layout analysis and verification tools. His research interests include design and analysis of VLSI design automation algorithms.

Sartaj Sahni (M'79–SM'86–F'88), for a photograph and a biography, please see p. 472 of this issue.