Efficient Algorithms For Polygon To Trapezoid Decomposition And Trapezoid Corner Stitching *

Qiao Li Sung-Mo (Steve) Kang

Computer & Systems Research Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL 61801

ABSTRACT

In non-Manhattan geometry layout extraction, polygon to trapezoid decomposition is an indispensable step. Its efficiency and the organization of generated trapezoids significantly affect the performance of layout extractors. We present a new polygon to trapezoid decomposition algorithm used in our layout extractor *iLEX*. The concept of *edge pair* and *scanline interval* are introduced to provide improved efficiency over conventional scanline algorithms. Definitions for trapezoid corner stitches are provided as well as integrated algorithms on corner stitching trapezoids generated. Complexity analysis shows that our scanline algorithm has an expected computation time of $O(n \log n)$, and an expected space of $O(\sqrt{n})$, where n is the number of non-vertical edges in the given layout.

Keywords

Non-Manhattan layout extraction, polygon to trapezoid decomposition, edge pair, scanline interval, scanline algorithm.

1. INTRODUCTION

For non-Manhattan geometry layout extraction, polygon to trapezoid decomposition is an indispensable step. Not only because trapezoids are more amenable to geometry operations such as neighbor finding, but also because a polygon in itself may not refer to a layout object at all. A polygon is defined by a sequence of points. Two adjacent points form an edge. The opaque side of the polygon lies to the left of an edge. Therefore, the donut shaped layout object in Fig. 1 (for example, a guard ring) can be represented by two polygons P_1 and P_2. Neither P_1 nor P_2 can be taken out of the context of each other.

*This research was supported in part by a grant from Semiconductor Research Corporation (SRC contract 98-DJ-613).

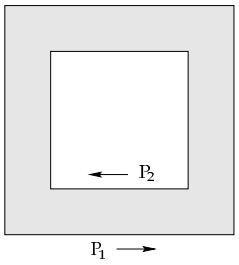

Figure 1: Polygons

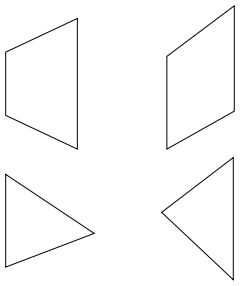

Figure 2: Trapezoids

A trapezoid (Fig. 2) is defined to consist of two parallel vertical edges, and two other edges which are not necessarily parallel. Only one of the two parallel vertical edges can degenerate into a point.

Edge based scanline algorithms [1; 3; 2; 7] have been extensively used in polygon to trapezoid decomposition as they are fast and memory efficient. Different from earlier scanline algorithms, our algorithm is *edge pair* based rather than edge based and through the use of *scanline intervals* much improves scanline processing, trapezoid detection and trapezoid generation. To facilitate later device extraction, we extended the definitions of corner stitches [6] to trapezoids, and provide algorithms on corner stitching trapezoids while they are generated.

Sec. 2 provides a review of earlier scanline algorithms. Sec. 3 and Sec. 4 detail our data structures as well as the scanline algorithm based on them. In Sec. 5 we extend definitions of corner stitches to trapezoids and present algorithms to corner stitch trapezoids while they are generated in our scanline algorithm. Finally, Sec. 6 and Sec. 7 provide complexity analysis and experimental results.

2. REVIEW AND MOTIVATION

In a scanline algorithm, a scanline is constructed to be swept on sorted polygon edges along x-axis from $-\infty$ to ∞. The scanline keeps only sorted records of edges being intersected to reduce memory requirement, and stops only at registered stop points to improve run time. At each stop point a list of events to be processed on the scanline is maintained. Events are caused by the need for processing of either new starting edges, or old ending edges, or crossing edges. Specific operations to be done at stop points are algorithm dependent.

In most earlier algorithms, at *each* stop point, one counter

for each side of the scanline has to be swept along the scanline to decide whether trapezoids have to be generated. This demands some delicate handling when more than 2 edges cross a point as in Fig. 3.

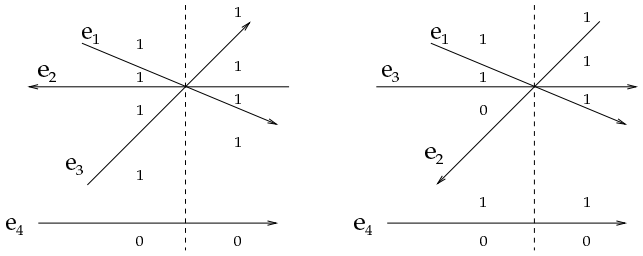

Figure 3: Trapezoid detection through counting

That sweeping has to be done at every stop point is because most scanline algorithms [3; 7] suffer from not knowing the ranges of effect of newly introduced edges, or newly terminated edges, or newly crossed edges. Chiang et al. [2] alleviated the problem by first decomposing individual polygon into trapezoids, and then feeding the generated trapezoids into the scanline algorithm. As decomposition is done on an individual polygon basis, overlapping trapezoids (at least from different polygons) have to be allowed. Also, before individual polygon can be decomposed into trapezoids, its edges have to be organized into TOP/BOTTOM pairs. All of them consume extra computation time.
It is proposed in [7] that only trapezoids on a band following the scanline are kept in memory to enable device extraction within the scanline algorithm. The same technique could be put into use in our scanline algorithm as well. Nevertheless, doing so will not only restrict devices extracted to be primitive, but also require significant modification of the scanline algorithm whenever a new device model is added to the extractor. We opt for an independent device extractor and a stand-alone scanline algorithm in which:

1. Rather than decomposing individual polygon into overlapping trapezoids and then working on the trapezoids to generate new trapezoids, we sweep the scanline from $-\infty$ to ∞ once, and generate non-overlapping trapezoids.

2. *Edge pair*s are used as input to the scanline algorithm. They provide the propagation *direction* and *range* of effect, and consequently no TOP/BOTTOM pairing information is needed.

3. Through the use of *scanline intervals*, we can detect when a trapezoid has to be generated and totally rid ourselves of sweeping counters along both sides of the scanline at every stop point.

4. Efficient algorithms on corner stitching generated trapezoids are integral parts of our scanline algorithm.

3. FUNDAMENTAL DATA STRUCTURES

Each pair of adjacent points v_i, v_{i+1} in a polygon representation $\{v_1, \ldots, v_n, v_{n+1}(= v_1)\}$ makes a directed edge $\{v_i, v_{i+1}\}$. As earlier scanline algorithms, only non-vertical edges are considered. An edge $\{(x_1, y_1), (x_2, y_2)\}$ is, `FORWARD` when $x_1 < x_2$, and `BACKWARD` when $x_1 > x_2$.

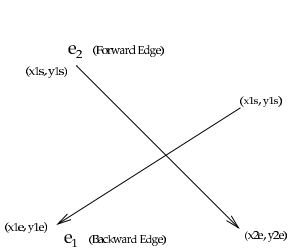

Figure 4: Edges

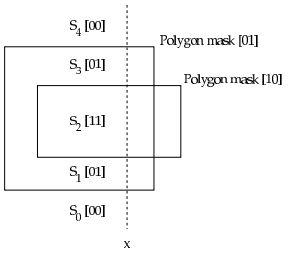

Figure 5: Scanline intervals

3.1 Edge Pair

It can be proved that each pair of adjacent edges [1] (e_1, e_2) of a polygon forms an edge pair from the following three mutually exclusive categories: starting edge pair, ending edge pair, and extending edge pair, as in Fig. 6. Adjacent edges e_1, e_2 form, starting edge pair when e_1 is `BACKWARD` and e_2 is `FORWARD`; ending edge pair when e_1 is `FORWARD` and e_2 is `BACKWARD`; extending edge pair when e_1 and e_2 are in the same direction. As their names imply, a starting edge

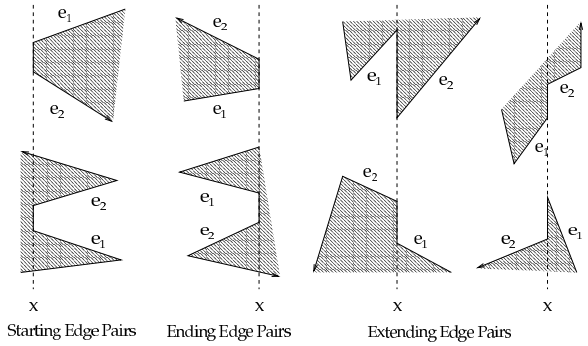

Figure 6: Edge Pairs

pair (e_1, e_2) specifies that e_1 and e_2 start from the same x-coordinate. An ending edge pair (e_1, e_2) similarly states that e_1 and e_2 end at same x-coordinate. An extending edge pair (e_1, e_2) says that e_2 starts from the same x-coordinate as e_1 ends. For example, for the first edge pair example given in Fig. 6, as e_2 follows e_1, differs from e_1 in direction, and the direction of e_1 is `BACKWARD`, (e_1, e_2) makes a starting edge pair.
Each non-vertical edge will be put into two edge pairs, one with the non-vertical edge ahead of it in the polygon representation, one with the one after it. For our purposes, edge direction is only used while constructing edge pairs. And once put into an edge pair, a `BACKWARD` edge e will swap its starting coordinate and its ending coordinate such that all edges will have starting x-coordinate smaller than ending x-coordinate.

3.2 Scanline Interval

Scanline intervals at stop point x are non-overlapping intervals formed by edges crossing the scanline at x (Fig. 5). With scanline intervals, trapezoid detection no longer requires sweeping of two counters along the scanline at each

[1] Edges separated by vertical edges only are adjacent as well as the first edge and the last edge of the polygon.

stop point, rather, as we shall see, it simply becomes a by-product of the processing for scanline intervals at each stop point.

Each scanline interval maintains its staring x-coordinate, upper/lower bounding edges, mask value, and for each layer, the number of polygons on it covering is kept so that layouts with overlapping polygons on the same layer can be handled correctly. Whenever a polygon on layer l begins to cover scanline interval s, the bit on the mask of s corresponding to layer l is set, and the number corresponding to l gets incremented. Whenever a polygon on layer l ceases to cover scanline interval s, the number corresponding to l is decremented, and when this number is 0, the corresponding bit on the mask is reset.

Shown in the next section, creation and deletion of a scanline interval are accomplished through split and merge operations introduced by edges. Intuitively, when an edge starts, it splits the scanline interval wherein its starting point resides. When an edge ends, two scanline intervals separated by it merge.

4. DECOMPOSITION ALGORITHMS

4.1 Edge Pairing

Earlier scanline algorithms suffer from *not* knowing the effect ranges of events to be processed at each stop point. Counters have to be swept along the scanline, as while preparing edges from polygons for scanline processing, much information implicit in the polygon representation is lost. From Fig. 6, we can see that an edge pair (e_1, e_2) disambiguously dictates how the polygon coverage of the polygon they belong to change before and after the x-coordinate they share, and the change is *confined* by e_1 and e_2. Thus, feeding edge pairs rather than edges to the scanline algorithm, we will know both the effect ranges of events and mask operation to be performed for all the scanline intervals within the effect range of an event.

Algorithm 1. *(Edge Pairing)*

```
1   Given adjacent edges (e1, e2)
2   if (e1 and e2 are in the same direction) then
3       if (e1 is a FORWARD edge) then
4           if (e2 above e1) then
5               Build extending edge pair (e1, e2, −)
6           else
7               Build extending edge pair (e1, e2, +)
8       else
9           if (e2 above e1) then
10              Build extending edge pair (e2, e1, −)
11          else
12              Build extending edge pair (e2, e1, +)
13  if (e1 is a BACKWARD edge) then
14      if (e1 below e2) then
15          Build starting edge pair (e1, e2, −)
16      else
17          Build starting edge pair (e2, e1, +)
18  else
19      if (e1 below e2) then
20          Build ending edge pair (e1, e2, −)
21      else
22          Build ending edge pair (e2, e1, +)
```

Each edge pair construction takes three parameters, lower and upper edges in the pair, and an action of either '+' or '−', specifying what operation will be performed on the polygon coverages of scanline intervals enclosed by those two edges when this edge pair is to be processed. For example, for the first starting edge pair in Fig. 6, as e_1 is above e_2, + is passed to the starting edge pair construction routine. It can be seen from the figure that polygon coverage to the right of the stop point is incremented comparing to the left of the stop point. Each kind of edge pair also has its own processing method,

Algorithm 2. *(Ending Edge Pair Processing)*

```
1   Given ending edge pair (e1, e2, a).
2   Update all scanline intervals between e1 and e2 with
    action a.
3   Merge scanline intervals separated by e1 and e2.
```

Algorithm 3. *(Starting Edge Pair Processing)*

```
1   Given starting edge pair (e1, e2, a).
2   Find scanline intervals s1, s2, where starting points
    of e1 and e2 reside.
3   Split s1 with e1, s2 with e2.
4   Insert new scanline intervals into the scanline.
5   Update all scanline intervals between e1 and e2 with
    action a.
```

Algorithm 4. *(Extending Edge Pair Processing)*

```
1   Given extending edge pair (e1, e2).
2   Find scanline intervals s2 where starting point of e2
    resides.
3   Split s2 with e2.
4   Update all scanline intervals between e1 and e2 with
    action a.
5   Merge scanline intervals separated by e1.
```

To make our scanline algorithm work for arbitrary layouts, many technical details have to be considered. In the following subsections, we present some of the significant issues. Others such as handling for edge crossing at ending points, handling for edges with common segments are omitted for bervity.

4.2 Crossing Edges Handling

To avoid computing unnecessary crossing points, only edges bounding the same scanline interval in the current scanline are checked for crossing points. Thus, only when either one of the bounding edges of a scanline interval changes, will a crossing point be calculated.

Before processing a point where edges cross, we construct a list of edges crossing it in top to bottom order from head to tail. Edges with higher slopes are at the tail while those with lower slopes are at the head. Notice that, after this stop point, the list of edges in top to bottom order will be exactly the reverse of the list before this stop point. Edges with lower slopes will be at the tail while those with higher slopes will be at the head. Meanwhile, scanline intervals affected by an edge e's crossing this point are those lower bounded by edges in the list before the processing between e's original position (exclusive) and e's position after the processing (inclusive). For example, in Fig. 7. The list before the processing in top to bottom order is $\{e_1, e_2, e_3\}$. The list after

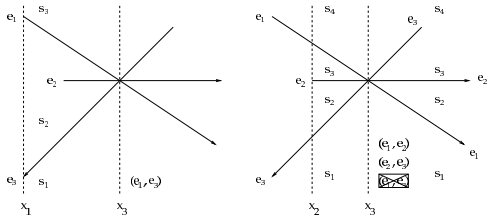

Figure 7: Changes of scanline intervals over crossing point

the processing in top to bottom order is $\{e_3, e_2, e_1\}$. Scanline intervals affected by e_1's crossing this point are those scanline intervals lower bounded by e_2 and e_3. As e_1 changes from element #1 to element #3, the scanline intervals affected are lower bounded by element #2 and element #3 in the list before the processing.

However, as scanline intervals split and merge, an edge may not remain a lower/upper bounding edge of one scanline interval all the time. Therefore it may register at one stop point as lower/upper crossing edge more than once. For example, edges e_1 and e_3 in Fig. 7, the crossing edge pair of (e_1, e_3) registered at stop point x_1 should be substituted by crossing edge pairs (e_1, e_2) and (e_2, e_3) at stop point x_2. Or alternatively, they can be easily ignored while constructing the descending slope edge list for edges crossing at this stop point.

Algorithm 5. *(Crossing Edges)*

```
1   Given n edges crossing the same point
2   Construct list of these n edges in ascending slope
3   for i ← 1 to ⌊n/2⌋ do
4       for j ← i + 1 to n − i do
5           if (e_i is a FORWARD edge) then
6               Increment coverage to scanline interval
                lower bounded by e_j.
7           else
8               Decrement coverage to scanline interval
                lower bounded by e_j.
9           if (e_{n−i} is a FORWARD edge) then
10              Decrement coverage to scanline interval
                lower bounded by e_j
11          else
12              Increment coverage to scanline interval
                lower bounded by e_j
13      done
14      e_i and e_{n−i} switch their positions in the list.
15      Calculate crossing points for scanline intervals
        with either e_i or e_{n−i} as a bounding edge.
16  done
```

4.3 Trapezoid Detection and Generation

During the processing of a stop point, new scanline intervals may be introduced and old scanline intervals may cease to exist. The following arrangements are made to tackle these cases:

1. Before processing at a stop point, preserve images of all scanline intervals [2].
2. *Newly created scanline interval* shares the image of the scanline interval it splits.
3. *Scanline interval ceasing to exist* checks its mask against that of its image before merges with another scanline interval.

Intuitively, trapezoid detection is accomplished by realizing that when the mask of a scanline interval changes from that of its image after a stop point processing, a trapezoid has to be generated. A good example of our processing arrangement is shown in Fig. 8 where all polygons are on the same layer. At x_2, e_2 spilts s_{12}, no trapezoid is to be generated, as after the split, s_{22} and s_{23} [3] are both covered by polygons just as s_{12}. At x_3, due to e_1, s_{21} and s_{22} merge into s_{31}, and s_{22} changes from with polygon coverage to s_{31} with no polygon coverage. Thus, a trapezoid has to be generated, and scanline interval image s_{22} of s_{31} is singled out as the seed for trapezoid generation.

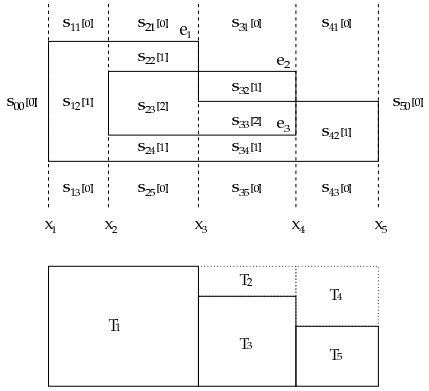

Figure 8: Trapezoid detection and generation

The above arrangements handle Manhattan geometries adequately. However, in case of crossing edges, more processing is needed. First of all, the scanline interval lower bounded by e after the processing has to take the image created for the scanline interval lower bounded by e before the processing. Second, a scanline interval is separated from its image by other scanline intervals and other scanline interval images. For example, in Fig. 7, s_3 after the processing is separated from its image by scanline interval s_4 and the image taken for s_4 before the processing [4]. Finally, it should be realized that only when there isn't an edge bisecting all scanline intervals and scanline interval images involved at this crossing point such that those above it have the same mask, and those below it has the same mask, do we need to generate trapezoids. As the algorithm handling this takes the points we made above literally, we omit its pseudo code here.

To minimize the number of trapezoids generated, each trapezoid generated is made as tall as possible. A trapezoid is generated from a sequence of scanline interval images sharing the same mask value if one of them has been picked out during the processing of this stop point as a seed for trapezoid generation.

[2] By making a scanline interval and its image share data, this image making requires minimal time and memory.

[3] Also s_{24} due to split from e_3 (starting pairing edge of e_2).

[4] Image taken for s_4 before the processing is the image for scanline interval s_2 after the processing, as they are both lower bounded by edge e_1.

Algorithm 6. *(Trapezoid Generation)*

1 Given sorted list of scanline interval images.
2 Generate trapezoids from seeded scanline intervals with neighboring same-mask scanline intervals.

Sorted scanline interval images are passed in step 1 to facilitate the integrated corner stitching algorithms presented later on. By passing an ascending list of scanline intervals, we can make sure that lower trapezoids are generated before upper trapezoids.

5. EXTENDING CORNER STITCHES TO TRAPEZOIDS

Marple et al. [4] extend corner stitch definitions [5] to trapezoids [5]. However, the extension is incorrect, as left and right neighbor iterations fail under certain circumstances. We define the corner stitches of a trapezoid T as,

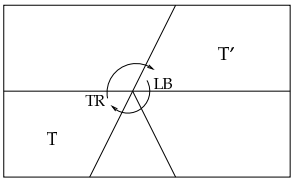

Figure 9: Marple's definition of corner stitches can result in incorrect left and right neighbor iterations.

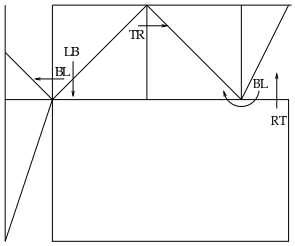

Figure 10: Our definition of corner stitches

`BL` points to the bottom-most trapezoid t overlapping T's left edge or containing lower left point of T if T and t are lower bounded by the same edge.

`LB` points to the left-most bottom trapezoid whose top edge shares a segment of T's bottom edge.

`TR` points to the top-most trapezoid t overlapping T's right edge or containing upper right point of T if T and t are upper bounded by the same edge.

`RT` points to the right-most top trapezoid whose bottom edge shares a segment of T's top edge.

Though the definitions are somewhat complicated. They are easily and intuitively integrated in our edge based scanline algorithm [6] , and take $O(1)$ time in average for each corner stitch assignment. We use $t \rightarrow ll$ to denote lower left point of t, $t \rightarrow ur$ upper right point of t.

5.1 Bottom-most Left Stitch - `BL`

As trapezoids are generated from left to right, `BL` of a trapezoid can be assigned by the time it is being generated. Each scanline interval maintains `BL` pointing to the appropriate trapezoid, and is updated as,

[5]In [4], trapezoids are defined as with 2 horizontal parallel edges, with either one of them can degenerate into a single point.

[6]With appropriate assignments of scanline interval images to scanline intervals, as well as edge merging. Omitted here for brevity.

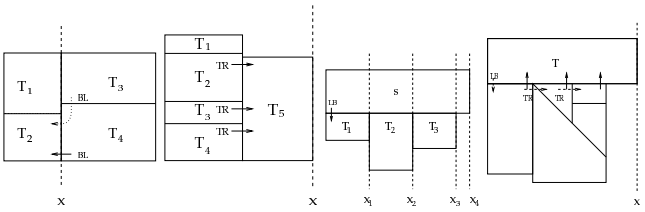

Figure 11: `BL` Figure 12: `TR` Figure 13: `LB` Figure 14: `RT`

1. *when scanline intervals merge*, `BL` of the merged scanline interval assumes the value of `BL` of the lower scanline interval.

2. *when a scanline interval splits*, `BL`'s of both generated scanline interval assume `BL` of the one being split.

This, nevertheless, may render `BL` not pointing a trapezoid lower than what it should be after several merges and splits (Fig. 11). However, we can prove that no matter what sequence of merges and splits a scanline interval undergoes, its `BL` will never be pointing to a trapezoid above the one it should, and thus, the right trapezoid its `BL` should be pointing to could be found by following the `RT`'s.

Algorithm 7. *(`BL` Assignment)*

1 Given bottom-most scanline interval s constituting a trapezoid
2 **while** ($s \rightarrow$`BL`) **do**
3 **if** ($s \rightarrow$`BL` contains $s \rightarrow ll$) **then**
4 **return** ($s \rightarrow$`BL`)
5 **else**
6 ($s \rightarrow$`BL`) $\leftarrow$ ($s \rightarrow$`BL`$\rightarrow$`RT`)
7 **done**

It is not possible for $s \rightarrow$`BL` to be `NIL` through `RT` following, as every point on the left edge of s must be on the right edge of a trapezoid – trapezoids just cannot enclose, and thus make a scanline interval landlocked.

5.2 Top-most Right Stitch - `TR`

By the time, a trapezoid is generated, there is no way we can tell what its `TR` is, as to the right of the scanline are scanline intervals not trapezoids. Thus, we have to rely on trapezoids generated to its right to notify it of its `TR`'s availability.

Algorithm 8. *(`TR` Assignment)*

1 Given trapezoid T just generated
2 $t \leftarrow (T \rightarrow$`BL`$)$
3 **while** (T contains $t \rightarrow ur$) **do**
4 ($t \rightarrow$`TR`$) \leftarrow T$
5 $t \leftarrow (t \rightarrow$`RT`$)$
6 **done**

For the same reason explain for `BL` tracking, it is not possible for t to be `NIL` through `RT` following as well.

5.3 Left-most Bottom Stitch - `LB`

As our scanline algorithm scans from left to right, as soon as the first trapezoid is generated below a scanline interval s, the trapezoid to be generated from this s *must* have that first trapezoid as `LB`.

Algorithm 9. *(`LB` Assignment)*

1 Given trapezoid T just generated, scanline interval s immediately above T
2 **if** (!($s \rightarrow$LB)) **then**
3 ($s \rightarrow$LB) $\leftarrow T$

From Fig. 13, we see that as soon as T_1 is generated, LB of s is assigned, and will not be affected by later generations of T_2 and T_3.

5.4 Right-most Top Stitch - RT

When a trapezoid T is generated, all trapezoids that should have T assigned as their RT have been generated. All we need to do is to go through lower neighbors of T and assign accordingly. As for trapezoids bordering T's lower edge yet haven't been generated yet, they shouldn't have T as their RT anyway, as they will have larger right x-coordinate than that of T.

Algorithm 10. *(RT Assignment)*

1 Given trapezoid T just generated.
2 $t \leftarrow (t \rightarrow$LB)
3 **while** (t) **do**
4 ($t \rightarrow$RT) $\leftarrow T$
5 $t \leftarrow (t \rightarrow$TR)
6 **done**

6. COMPLEXITY ANALYSES

6.1 Polygon to Trapezoid Decomposition

Edge pairing takes $O(n)$ time, as it processes each edge in the layout only once and generates $O(n)$ edge pairs. Processing of an starting edge pair or extending edge pair takes $O(\log n)$ time, as scanline intervals are created. Processing of an ending edge pair, however, takes $O(1)$, as only scanline interval merging operation is carried out. For an average layout, the number of edges crossing at one point is $O(1)$. Thus crossing edges processing at any point will take $O(1)$ time. As in the average case [3; 7], at any time, the number of scanline intervals is $O(\sqrt{n})$, and there are $O(\sqrt{n})$ stop points, we have at most $O(n)$ crossing points. To sum up, our scanline algorithm has a total complexity of $O(n \log n)$.

6.2 Trapezoid Corner Stitching

Each of the corner stitching algorithm is called only once for each of the trapezoids generated. As in the average case at any time, the number of scanline intervals is $O(\sqrt{n})$, and there are $O(\sqrt{n})$ stop points, we have $O(n)$ trapezoids. There are four corner stitches for each trapezoid. For every corner stitch other than BL, every operation is an assignment for one *not* previously assigned corner stitch. Operations in BL assignment are only wasted on split scanline intervals caused by edges, and there only $O(n)$ edges! Consequently, BL operation takes $O(n)$ time as well. Thus corner stitching algorithms have a time complexity of $O(n)$.
From trapezoid corner stitching algorithms, we see that only the band of trapezoids adjacent to current scanline intervals need to be kept in memory. The expected number of these resident trapezoids at any time is $O(\sqrt{n})$, the same as the number of scanline intervals at any time. Thus space complexity of our scanline algorithm with integrated corner stitching algorithms is $O(\sqrt{n})$.

Table 1: Number of trapezoids generated (N), time (T), and memory (M) usage of our scanline algorithm

Circuit	Description	N	T(s)	M(MB)
c1	self	5,630	2.75	0.304
c2	c1 $\frac{1}{4}$ overlap c1	10,095	7.67	0.524
c3	c1 $\frac{1}{2}$ overlap c1	11,173	8.05	0.576
c4	c1 slightly off c1	12,149	9.66	0.700

7. EXPERIMENTAL RESULTS

All algorithms given in the paper have been implemented in roughly 2,000 comment-free C lines. Table 1 shows the time/memory used by our algorithms for various layouts. The testing environment is unoptimized gcc-2.8.1 compiled code running on Sun Ultra-2 with 256MB of memory.
We take c1, a 45^o clockwise rotated copy of a tri-state output pad layout with 14 layers and 2,016 polygons, and overlap copies of it onto itself so as to simulate fragmentation. Indeed the processing time and memory usage increase with increasing fragmentation. The experimental results corroborate our complexity analysis.

8. CONCLUSION

In summary, we provided a general frame work for polygon to trapezoid decomposition. Our algorithms are more intuitive and more efficient than earlier algorithms because of the two data structures, edge pair and scanline interval, introduced.

9. REFERENCES

[1] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computing*, 6-28(9):643–647, September 1979.

[2] K. W. Chiang, S. Nahar, and C. Y. Lo. Time-efficient vlsi artwork analysis algorithms in goalie2. *IEEE Transactions on Computer-Aided Design*, 8(6):640–648, June 1989.

[3] U. Lauther. An o(nlogn) algorithm for boolean mask operations. In *18th Design Automation Conference*, pages 555–562, 1981.

[4] D. Marple, M. Smulders, and H. Hegen. Tailor: A layout system based on trapezoidal corner stitching. *IEEE Transaction on Computer-Aided Design*, 9(1):66–90, January 1990.

[5] J. K. Ousterhout. Corner stitching: A data-structuring technique for vlsi layout tools. *IEEE Transactions on Computer-Aided Design*, CAD-3(1):87–100, January 1984.

[6] W. S. Scott and J. K. Ousterhout. Magic's circuit extractor. In *22nd Design Automation Conference*, pages 286–292, 1985.

[7] N. P. van der Meijs and A. J. van Genderen. An efficient algorithm for analysis of non-orthogonal layout. In *IEEE International Symposium on Circuits and Systems*, pages 47–52, 1989.