A Fast, Practical Algorithm for the Trapezoidation of Simple Polygons

Thomas F. Hain, David D. Langan, III

Abstract— *A fast, practical, deterministic algorithm for the horizontal trapezoidation of simple polygons is presented. The polygon is decomposed into a minimal collection of trapezoid sequences, such that two trapezoids adjacent within a sequence always share a common horizontal border. Such trapezoid sequences are a convenient data structure in a display list for a collection of polygonal objects to be filled/rendered. A linear traversal of the polygon outline identifies a subset of critical vertices, which are then processed in sweep order. Horizontal as well as vertical edges—very common in practical polygons—are handled explicitly. Complexity is linear for most 'practical' polygons, with non-linear running times being required only for much less frequently occurring geometries. A straightforward extension allows trapezoidation of simple polygons with holes.*

Index Terms—Computational geometry, rendering (computer graphics), trapezoidation, polygon decomposition.

I. Introduction

Decomposing a simple polygon has been one of the most challenging problems in two-dimensional computational geometry. It is a basic primitive in computer graphics and, generally seems the natural preprocessing step for most nontrivial operations on simple polygons [[4],[3]]. The goal of polygonal decomposition is to break a given simple planar polygon into a set of standard two-dimensional shapes (most often non-overlapping triangles or trapezoids) without adding any new vertices. Decompositions into trapezoids and triangles are equivalent in terms of computational complexity, since one problem can be reduced to the other in linear time [[7]]. Decompositions are often used to reduce problems involving complicated regions to problems, involving primitive shapes, that are generally easier to solve.

Definition: A simple polygon is the region of a plane bounded by a finite collection of line segments forming a simple closed curve. Let $v_0, v_1, \cdots, v_{n-1}$ be n points in the plane with a cyclic ordering. Let $e_0 = (v_0, v_1), \cdots, e_i = (v_i, v_{i+1}), e_{n-1}$ be n edges connecting the points (vertices). Then these edges bound a polygon iff [[17]]

1. The intersection of each pair of edges adjacent in the cyclic ordering is the single vertex shared between them: $e_i \cap e_{i+1} = v_{i+1}$, for all $i = 0, \cdots, n-1$.
2. Nonadjacent edges do not intersect: $e_i \cap e_j = \phi$, for all $j \neq i+1$.

In particular, the storing and rendering of filled polygons in computer graphics often requires such a decomposition. In the case of trapezoidation, it is further convenient to organize the trapezoids into a minimal collection of 'trapezoid sequences' so that common edges of adjacent trapezoids can be represented more compactly. In fact, each trapezoid sequence covers a monotone polygon component of the general simple polygon. Since monotone polygons are easy to resolve into trapezoids [[17]], the general trapezoidation problem could be approached as one of decomposing a general polygon into monotone polygons, which are then further decomposed into the constituent trapezoids. This is the approach taken here, with the output being a collection of horizontal trapezoids (partitioned into a set of trapezoid sequences.)

Definition: A *trapezoid sequence* is an ordered list of horizontal trapezoids $(t_0, t_1, \cdots, t_{k-1})$ such that $\{(y_i^{bottom} = y_{i+1}^{top}) \wedge (x_i^{bottom} \cap x_{i+1}^{top} \neq \phi) \mid i = 0, \cdots, k-2\}$, where y_i^{bottom} is the y-coordinate of the bottom of the ith trapezoid, and x_i^{bottom} is the range of x-coordinates covered by the bottom edge of the ith trapezoid. That is, the bottom edge of a trapezoid overlaps the top edge of the next trapezoid.

Definition: A *horizontal decomposition* of a polygon into trapezoid sequences is a set of trapezoid sequences such the union of all trapezoids in all trapezoid sequences is the trapezoidation of the polygon. That is, the horizontal decomposition is a (generally non-unique) partitioning of trapezoids into a collection of trapezoid sequences.

As an example of the need for a fast algorithm for the trapezoidation of polygons, consider the field of printer graphics. A printed page contains either geometric filled shapes or bitmaps. All geometrically defined shapes are typically specified by areas bounded by conic, Bézier, or straight-line segments. Curves are then approximated to a series of straight-line segments by a process called "flattening" the curve. Thus, geometric shapes to be filled are almost always described by polygons. The polygons that ultimately will constitute a printed page are then converted into trapezoid sequences that are stored in a display list. This display list is a data structure, which is a spatially compressed representation of the page that

Manuscript received March 4, 2005.

T. F. Hain, and D. D. Langan, are with the School of Computer & Information Sciences, University of South Alabama, Mobile, AL 36688 USA (phone: 251-460-6390; e-mail: thain@usouthal.edu, dlangan@usouthal.edu).

can be efficiently stored, reused, and rendered. It is clear that the speed of the trapezoidation algorithm will have an impact on printer page output rates. Currently, filling polygons is typically done scan-line by scan-line using an active edge table to keep track of scan-line spans interior to the polygon. This is an image precision algorithm, and depends on the specific resolution of the device, and requires a great deal of processing, specially in the case of high-resolution display devices where a polygon can span a large number of scan lines. Object precision algorithms, such as the algorithm described here, have the advantage of being device resolution independent.

Section II of this paper gives a brief review of existing approaches to this problem while section III describes the algorithm. Section IV gives a proof outline, and section V gives a complexity analysis. Section VI illustrates the used implementation approach, while Section VII gives comparative performance results.

II. Previous work

Quadratic triangulation algorithms have been implicit in proofs since Lennes' 1911 paper [[14]]. In 1978, Garey, Johnson, Preparata, and Tarjan [[8]] triangulated a simple polygon by decomposing it into monotone polygons and then decomposing each of these separately into triangles using an algorithm also described in that paper. The decomposition into monotone polygons uses the regularization procedure introduced by Lee and Preparata [[13]]. This procedure has $O(n \log n)$ complexity, where n is the number of vertices, and adds to the polygon non-intersecting diagonals, which do not cross the polygon boundary.

A 1982 algorithm by Chazelle [[1]] is particularly easy to implement. It finds a diagonal of the polygon P that divides it into two polygons P_1 and P_2, such that the number of vertices in P_1 and P_2 are each less than $\frac{2}{3}|P|+2$. The algorithm finds such a diagonal in $O(|P|)$ time. A simple divide-and-conquer algorithm based on this technique yields a total complexity of $O(n \log n)$.

A 1983 paper by Hertel and Mehlhorn [[10]] combined the steps of the algorithm of Garey et al. [[8]] into one sweep to yield an $O(n \log n)$ algorithm. They then improved it to $O(n \log r)$, where r is the number of concave angles (internal angles >180°) within the input polygon, by restricting the sweep event points to only $O(r)$ points of the polygon. The sweep line also is no longer a simple straight line. A crooked line is varied, since some points do not get processed until the actual sweep line is well past them. This algorithm works even if the simple input polygon has polygonal holes.

Like that of Hertel and Melhorn [[10]], Chazelle and Incerpi's 1984 algorithm [[5]] depends on the geometric complexity of the polygon. Any simple polygon can be decomposed into alternating sequences of clockwise and counterclockwise spiraling chains: they define *sinuosity, s,* as the number of such chains. The complexity of their algorithm is $O(n \log s)$.

All attempts to start with the 'monotonization' or 'diagonal splitting' path failed until 1984, when Fournier and Montuno [[7]] showed the equivalence of trapezoidal decomposition with triangulation for simple polygons, and as a result, many recent efforts have been concentrated on polygon trapezoidation. Their deterministic sweep algorithm constructs vertical trapezoids defined by two non-adjacent vertices of the input polygon, P, in time $O(n \log n)$. Once trapezoidalized, P is broken into monotone polygons and triangulated by joining the trapezoid defining vertices.

Clarkson, along with Tarjan [[6]] in 1989 devised a randomized algorithm based on the divide-and-conquer strategy that finds the visibility partition of the polygons with respect to a random subset of edges. The polygon is then recursively subdivided about that partition into smaller polygons using Jordan sorting. The algorithm runs in expected $O(n \log^* n)$. It can also be used to check whether or not a given polygon is simple.

In 1990, Kirkpatrick, Klawe, and Tarjan [[11]] presented an $O(n \log \log n)$ algorithm employing much simpler data structures. In addition, their algorithm can be modified to run in $O(n \log^* n)$ if the coordinates of the points of the polygon are integers bounded by a fixed polynomial in n. The basic idea is the same as in Tarjan's 1988 paper, but the triangle splitting is achieved without any Jordan sorting or other complicated data structures. For bounded integer coordinates, this algorithm builds a data structure in linear time that can answer queries about horizontal neighbors in $O(\log^2 n)$.

In 1991, Seidel [[19]] introduced a randomized incremental algorithm to compute the horizontal visibility map (trapezoidation) of a simple polygon. It has a complexity $O(n \log^* n)$, but is much simpler to implement than that of Clarkson and Tarjan. In fact, he asserts that its implementation simplicity is "a property that very few, if any, of the algorithms mentioned can claim."

Also in 1991, Bernard Chazelle [[3]] discovered a linear time, deterministic algorithm that settled the question about the intrinsic computational complexity of triangulation once and for all. However, the algorithm is according to Toussaint [[21]] "unimplementable," and according to O'Rourke [[17]] contains "details [that] are formidable." This algorithm has, to the authors' knowledge, never been implemented.

III. Current trapezoidation algorithm

The presented algorithm has a precondition that the polygon is simple, and (in the current description) that the vertices are given in clockwise order around the polygon starting at some arbitrary vertex. It will also be assumed that unnecessary inline vertices have been previously removed by a simple, linear, preprocessing stage.

Vertex coordinates are snapped to an arbitrarily fine (within the representation) grid to allow direct coordinate comparison without having to worry about round-offs.

A. Preliminaries

To help in the discussion of the algorithm, some terminology is now defined:

Definition: A *chain* $C = (v_1, v_2, \cdots, v_p)$ is a planar straight-line graph with vertex set $\{v_1, v_2, \cdots, v_p\}$ and edge set $\{(v_i, v_{i+1}) \mid i = 1, \cdots, p-1\}$. [[18]]

Definition: A chain $C = (v_1, v_2, \cdots, v_p)$ is said to be *monotone* relative to a given line *l* if a line orthogonal to *l* intersects *C* at exactly one point. That is, the orthogonal projections $\{l(v_1), l(v_2), \cdots l(v_p)\}$ of the vertices of *C* on *l* are ordered as $l(v_1), l(v_2), \cdots, l(v_p)$. [[18]] We extend the ordering to be nonstrict—using either the ≤ or ≥ relations.

Definition: A *falling chain* is a maximal (in the sense that it includes any horizontal edges at either end) chain that is monotone relative to a vertical line, and having non-increasing *y*-coordinates. A falling chain has the property DOWNTORIGHT if the lower end is to the right of the upper end. A *rising chain* is non-maximal, but is otherwise similarly defined.

Definition: A *polygon* is *monotone* if its boundary is composed of exactly two non-intersecting monotone chains relative to the same line. For example, a polygon is vertically monotone if its boundary is composed of two vertically monotone chains: the polygon's left chain and right chain. In this case, each chain terminates at the polygon's uppermost vertex (assuming no horizontal edges at the top and bottom) and lowermost vertex and contains zero or more vertices in between. [[12]]

Definition: A falling chain is split into two *down-chains* at the rightmost vertex (with ties being broken by taking the lower vertex of a vertical edge). If the right extremum is at either end of the falling chain, one of the down-chains is null.

Definition: Similarly, a rising chain is split into two up-chains at the leftmost vertex (with ties being broken by taking the upper vertex of a vertical edge). Again, if the left extremum is at either end of the rising chain, one of the up-chains is null.

Definition: A *join vertex*, or simply a *join*, is a vertex that connects (i.e., is an element of adjacent) up-chains and/or down-chains. That is, a join vertex $v_i = C^k_{i-n+1,i} \cap C^{k+1}_{i,i+m-1}$, where C^k and C^{k+1} are two successive up[down] chains of length *n* and *m* successively.

There are 10 distinct join types, and are illustrated in Figure 1, where the heavy directed arcs represent up-chains, the lighter directed arcs represent down-chains, and the join is the connecting vertex. The shading represents the side of the chains interior to the polygon (which you will recall is clockwise oriented).

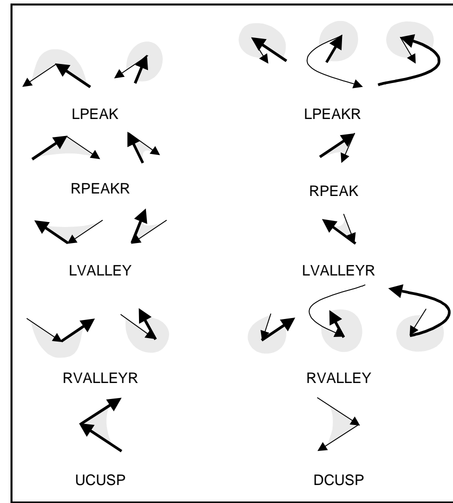

Figure 1 Join Types. (Arcs represent chains)

The join types are defined in Table 1. It should be noted that, because of the precondition for no inline vertices, the condition $\vec{e}_i \times \vec{e}_{i+1} = 0$ cannot occur.

The algorithm proceeds by identifying all *j* join vertices in a pass through the vertex list. It can be readily seen that, for practical[1] polygons, $j << n$, where *n* is the number of vertices in the polygon, and *j* is the number of join vertices. The join vertices are then sorted lexicographically by their *x*-coordinate, and negative *y*-coordinate. While this sorting could be done by a Jordon sort in linear time, it is more efficient to use a quicksort for practical polygons of bounded size. The algorithm then processes (i.e., sweeps) joins sequentially from left to right. Trapezoid (sequences) are generated during the processing of certain join types (i.e., at certain sweep events.)

Vertices that are not joins—that are internal to chains—will generally only be visited twice, once during join identification, and once during trapezoid generation. At any point in the join processing (i.e., at a sweep event), there will exist one or more windows, which are composed of concatenated fragments of up-chains. These chains are what can be seen looking toward the left from any point having the current sweep event's *x*-coordinate, and therefore represent a potential sequence of left sides for adjacent trapezoids.

Definition: A *window* relative to join *p*, is a transient (existing at sweep event *p*) up-chain, and is composed of fragments of windows, connected by horizontal segments. Four operations exist on windows:

1. Creation from an up-chain.
2. Splitting at a y-interpolation (see below).
3. Extension, by concatenating an up-chain.
4. Reduction (possibly completely), during trapezoid generation.

[1] Practical polygons are those defining typical graphical objects outlined by flattened curves and straight line segments. In these cases, the length of chains tends not to be as small as, for example, random generated and non-smoothed polygons.

Table 1 Definition of Join Types.

Join type	Definition
LPEAK	Vertex, v_i, joining top of up-chain and a non-DOWNTORIGHT down-chain, such that the vector product of the incident edges, $\vec{e}_i \times \vec{e}_{i+1} > 0$.
LPEAKR	Vertex, v_i, joining top of up-chain and a DOWNTORIGHT down-chain, such that the vector product of the incident edges, $\vec{e}_i \times \vec{e}_{i+1} > 0$.
RPEAK	Vertex, v_i, joining top of up-chain and a non-DOWNTORIGHT down-chain, such that the vector product of the incident edges, $\vec{e}_i \times \vec{e}_{i+1} < 0$.
RPEAKR	Vertex, v_i, joining top of up-chain and a DOWNTORIGHT down-chain, such that the vector product of the incident edges, $\vec{e}_i \times \vec{e}_{i+1} < 0$.
LVALLEY	Vertex, v_i, joining bottom of up-chain and a non-DOWNTORIGHT down-chain, such that the vector product of the incident edges, $\vec{e}_i \times \vec{e}_{i+1} < 0$.
LVALLEYR	Vertex, v_i, joining bottom of up-chain and a DOWNTORIGHT down-chain, such that the vector product of the incident edges, $\vec{e}_i \times \vec{e}_{i+1} < 0$.
RVALLEY	Vertex, v_i, joining bottom of up-chain and a non-DOWNTORIGHT down-chain, such that the vector product of the incident edges, $\vec{e}_i \times \vec{e}_{i+1} > 0$.
RVALLEYR	Vertex, v_i, joining bottom of up-chain and a DOWNTORIGHT down-chain, such that the vector product of the incident edges, $\vec{e}_i \times \vec{e}_{i+1} > 0$.
UCUSP	Vertex, v_i, joining two up-chains.
DCUSP	Vertex, v_i, joining two down-chains.

Windows are sequentially created, evolve, and are used up during join processing. The concept and role of windows are illustrated in Figure 2. The edges of this graph represent the up-chains and down-chains, and the nodes represent the joins, which are labeled in sweep order. The bold edges in Figure 2(a), (b), and (c) represent the existing windows just prior to processing joins 3, 6, and 7 respectively.

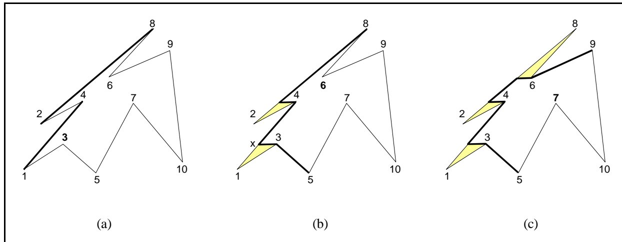

Figure 2 Example of windows.

After processing join 3, the window from join 1 to 4—denoted (1, 4)—will be split at its y-interpolation at the height of join 3 (point X). Now the lower window will consist of the up-chain (5,3), a horizontal segment from join 3 to X, and the upper part of the split window, (X,4). The "ear" bounded by the down-chain (3,1), the bottom part of the split window, and the horizontal edge from X to join 3, is a vertically monotone polygon, and will be converted to trapezoids, and taken out of consideration for the remainder of the process.

Definition: A *y-interpolation* at a height y of a window W exists if the window vertically spans y, and, in this case, is one of three types. It is the point having y-coordinate y, and x-coordinate determined for each of three types as follows:

Type I: If there exist two vertices in W at height y (i.e., a horizontal edge at height y): the x-coordinate of the vertex incident on the head end of the horizontal edge. We will call the vertex at the tail end of the horizontal edge the *secondary y-interpolation.*

Type II: If there exists a single vertex in W at height y: the x-coordinate of that vertex.

Type III: If there exist no vertices in W at height y: the x-coordinate of the intersection of the horizontal line at height y and the edge in W that vertically spans y.

The window y-interpolation types are illustrated in Figure 3. The directed arcs are clockwise-specified edges in the window. All points labeled p are y-interpolations at the height indicated by the dashed horizontal line. For type I interpolations, points labeled p' are secondary y-interpolations.

Definition: In any polygon (or polygon fragment) there is one window W relative to the current join p (sweep event), that is distinguished by having the largest (rightmost) y-interpolation at height y_p to the left of p. Window, W, is termed *left-facing* join p.

B. Join Processing

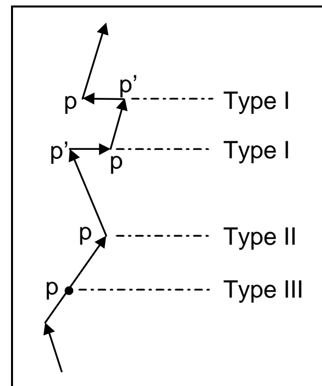

Figure 3 Window y-interpolation types

The processing of each type of join will now be described. For each of the peak and valley joins, there will be exactly one incident up-chain. If this up-chain is seen for the first time (i.e., the sweep event join is at the left end of the chain), it is marked as a window by labeling the top join of the chain. If the sweep event join is at the right end of the up-chain, it does no harm to label the chain as a window, since it will already be labeled as such. Thus, in any case, the top of the incident up-chain is marked as the top of a window.

The specific (additional) processing required by each join type will now be described. The initial pass over the vertices of the polygon has already provided explicit clock-wise linkages between joins/nodes. The terms 'prior' and 'next' when applied to joins will refer to those linkages.

1) ucusp

A UCUSP vertex is the leftmost vertex of a rising chain, and is thus the earliest point in a sweep when the rising chain will be seen. The two up-chains on either side of the join are linked into a single window. This ensures that all windows to the left of a join will exist (or will have been used up during trapezoidation) before that join is processed. Given explicit linkages between successive joins, the processing of this type of join is done in constant time.

2) dcusp

By the time this join is processed, it will be the rightmost vertex of a (remaining) monotone polygonal component. The left side of this polygon is the window incident on the top join, which is the join prior to the dcusp join itself. The right side of the polygon is the falling chain containing the dcusp join, i.e., the concatenation of the two down-chains connected by the dcusp join. This monotone polygon is simply reduced to a trapezoid sequence. Given the explicit linkage between adjacent joins, as well as the explicit linkage between vertices (existing or that are created during interpolation), the processing of this type of join is done in a time that is linear in the number of vertices in the monotone polygonal component being reduced.

3) rpeakr (and similarly lvalley)

No processing, (other than marking the incident up-chain as a window), is required. The processing of this type of join requires constant time.

4) rpeak (AND SIMILARLY lvalleyr)

As in the dcusp case, this is the rightmost vertex of a remaining monotone polygonal component. The left side of this polygon is the window incident on this join, and the right side is the down-chain incident on the same join. This monotone polygon is simply reduced to a trapezoid sequence. Again, given the explicit linkage between vertices , the processing of this type of join is done in a time that is linear in the number of vertices in the monotone polygonal component being reduced.

5) rvalleyr (and similarly lpeak)

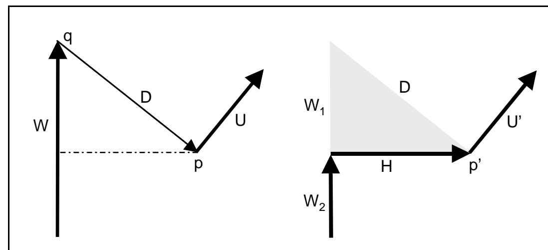

Figure 4 rvalleyr join handling.

The handling of rvalleyr joins is illustrated in Figure 4,where vertex p is the current (rvalleyr) join, and U and D are respectively the up-chain and down-chain incident on p. The window, W, incident on q, the join prior to p (i.e., the top of D), is split at the y-interpolation of W at the height of p. A horizontal edge H is defined from this interpolation (or the secondary y-interpolation, if it is of Type I) to p' (equivalent to p or to the next vertex if there exists a horizontal edge incident on p). The lower part of the split window, W2 (up to the y-interpolation of W at yp, or, if it is of Type I, the secondary interpolation), is connected via H to the up-chain, U' (equivalent to U with the initial horizontal edge removed, if it exists), forming an updated window. The removed shaded "ear" is a vertically monotone polygon whose left chain is the top part of the split window, W1, whose right chain is the down-chain D, incident on p, and whose bottom is closed by a horizontal edge from p to the y-interpolation of W at height yp. This monotone polygon is simply reduced to a trapezoid sequence. Again, assuming explicit linkages exist between adjacent edges, it can easily be seen that the processing of this type of join can be done in time that is linear in the number of vertices in the trapezoid sequence being reduced.

6) rvalley (AND SIMILARLY lpeakr)

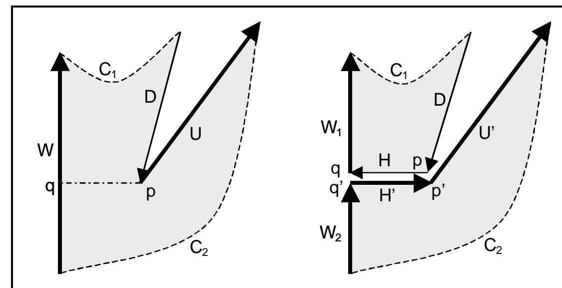

Figure 5 rvalley join handling.

We will call join vertices of type RVALLEY and LPEAKR left spike joins. The handling of RVALLEY joins is illustrated in Figure 5, where vertex *p* is the current (RVALLEY) join. Chains *U* and *D* are respectively the up-chain and down-chain incident on *p*. First we must search (by a method discussed below) for the window, *W*, that is left-facing *p*. The *y*-interpolation of *W* at height y_p is the point *q*. The dashed arcs C_1 and C_2 represent the series of chains/windows connecting the top and bottom of *W* respectively to the ends of *D* and *U* opposite to *p*. Point q' is equivalent to either *q* or, in the case of type I interpolations, the secondary *y*-interpolation of *W* at height y_p. Point p' is equivalent to either *p* or, if there exists a horizontal edge incident on *p*, to the next vertex. Chain U' is equivalent to *U* with any initial horizontal edge removed. Horizontal edge *H'* is defined from q' to p', and *H* is defined from *p* to *q*. Thus the polygon is split into two disconnected polygons defined by $[H, W_1, C_1, D]$ and $[H', U', C_2, W_2]$.

C. Searching for interpolation window

In the left spike join cases (RVALLEY and LPEAKR), a search is required for the specific window that is left-facing the spike join. In one approach, the search begins at the join prior to the spike join, and traverses join vertices in a counter-clockwise order, looking for windows, which vertically span the join

(which will be termed *candidate windows*). Each such candidate window is interpolated and either eliminated (its *y*-interpolation is to the left of the *y*-interpolation of a previous potential left-facing window, or is to the right of the join), or otherwise accepted as the new potential left-facing window.

Figure 6(a) illustrates this process for spike join 5. As joins are traversed counter-clockwise from this join, the first candidate window (i.e., one that meets vertical spanning criterion) is *A*. After traversing further, candidate window *B* is discovered with an interpolation closer and to the left of vertex 5, and becomes the new potential left-facing window. Candidate window *C* is found thereafter, with a closer interpolation on the left of the left spike join. The search is terminated when the join traversal returns to the spike join. An optimization to potentially reduce the number of required interpolations can be fairly easily implemented.

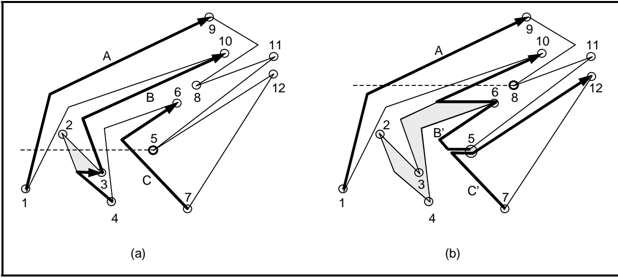

Figure 6 Search for window in spike case.

IV. Proof Outline

All left sides of horizontal trapezoids will be part of a window (as defined in section III), i.e., composed of (fragments of) up-chain edges. Similarly, the right sides of trapezoids will be composed of (fragments of) down-chain edges. This can be seen by walking the contour of the polygon (in the prescribed clockwise direction); the inside of the (simple) polygon, and consequently the component trapezoids, will always be on the right-hand side. The rightmost end, join *r*, of a down-chain, *D*, will trigger the trapezoidation of the (monotone polygonal) area between *D*, and the horizontally projected portion, *W'*, of a unique window, *W*, that exists at the sweep event *r*. *W* is the window left-facing *r*, and must exist at the sweep event *r* since up-chains are converted to, or incorporated into, windows as soon as they are seen (i.e., as soon as the leftmost end of an up-chain is the current sweep event). Every point on *D* must have the same unique left-facing window, *W*, that is $W' \subseteq W$, since both *W* and *D* are vertically monotone. Both *D* and *W'* will be consumed in the trapezoidation process for this component of the polygon. Any remnant window $W - W'$ will be extended to form the basis for a new window at a later sweep event.

V. Complexity Analysis

In this analysis we assume that all vertices (including vertices created by window interpolations) are doubly linked in order around the polygon (or polygon component). We also assume that join vertices are doubly linked in order around the polygon. The initial set-up of such linkages can be done in linear time by a single pass through the vertex list of the original polygon.

Joins of type UCUSP, RPEAKR, and LVALLEY can be processed in constant time. The complete set of trapezoids for a given input polygon is generated disjointly during the processing of joins of type RPEAK, LVALLEYR, DCUSP, RVALLEYR, or LPEAK. These five join types individually incur constant processing time, and together they incur time for the generation of $n-1$ trapezoids in the worst case (each time a pair of vertices are in horizontal view of each, or there exists a horizontal edge, the trapezoid count decreases by one). Thus the total time to process these eight join types is linear in *n*, the number of vertices in the input polygon.

The processing of spike joins requires the traversal of all join vertices in the polygon fragment that contains the spike. At least one window will meet the requirements given in section III.B.6), and this (left-facing) window will require, on average, the traversal of half its vertices. The total number of vertices visited has an upper bound of the number of vertices in the component, with the initial fragment size being *n*. Each window splitting event divides the polygon into two polygonal components. If we assume that the components are distributed randomly, the complexity of treating spike joins will be bounded by $\mathrm{O}(n \log n)$. In practical cases, the number of joins, *j*, will be relatively small, that is $j << n$. We know that $j < s$, where *s* is the sinuousity of the polygon, since changes in direction having a vertical component create peaks and valleys (i.e., joins), and only some of the remaining changes in direction (which have a horizontal component) create UCUSPS and DCUSP joins. Thus, a tighter bound is $\mathrm{O}(s \log n)$. If we further assert, as discussed in III.C, for practical inputs, that the number of windows that need to be interpolated is on average very small (close to 1), then a tighter estimate of the average-case complexity of join processing time is $\mathrm{O}(j_s \log j)$, where $j_s < j$ is the number of spike joins. For random polygons, $\langle j_s \rangle \approx j/16$ since there are two spike join types, with the down-chain direction constrained to one quadrant, and the up-chain direction constrained to one subdivision (on average half) of that quadrant. It is believed that j_s will be less than that in more practical polygons, since spike joins are somewhat 'unnatural'.

Thus, with the exception of spike joins, which are expected to be fairly rare in practical polygons, the algorithm is linear. It is believed that spike joins themselves are not very expensive in practical polygons.

VI. Implementation

In the following, an implementation involving a relatively simple linked data structure is described. The nodes in this

structure initially come from the vertices of the input polygon, and temporary nodes corresponding to edge interpolations may be added during the join processing. The suggested approach has been coded and tested, and the performance is shown in the following section.

An initial pass is made though the vertex list of the input polygon to create a doubly linked circular list of vertices. During this pass, join vertices that connect up-chains and down-chains are identified, labeled, and linked (by a separate thread) to form a doubly linked circular list. The peak and valley joins are labeled using a 3-bit code where separate bits represent characteristics of positive edge cross product, `downtoright`, and `peak`.

A rising chain that has an x-extremum strictly to the left of the top and bottom joins is split into two up-chains at this vertex. This vertex becomes a `ucusp` join (connecting the two resulting up-chains) and is linked into the join list. Similarly, a falling chain that has an x-extremum strictly to the right of the top and bottom joins, is split into two down-chains at this vertex. This `dcusp` vertex is also linked into the join list. Thus the vertices are doubly linked to adjacent vertices, and, if they are joins, to prior and next joins. Cusps use two additional bits in the join type code.

Additional nodes, corresponding to edge interpolations, are inserted as windows are y-interpolated. The circular lists are split into two disjoint circular lists, corresponding to polygonal components, during the processing of `lpeakr`, or `rvalley` (spike) joins. Sections of a list are removed (and transformed into trapezoid sequences) during the processing of `rvalleyr`, or `lpeak` joins. Circularly closed lists are completely transformed into trapezoid sequences during the processing of `dcusp`, `rpeak`, or `lvalleyr` joins. This dynamic multi-threaded list is the only data structure used, other than a static queue of joins to be processed.

VII. Performance Measurement

Approximately 600,000 polygons were extracted from a large suite of vector graphics test pages provided by QMS, Inc. This provided a “practical” dataset, on which relative timings were performed on the current implementation, in relation to Narkhede’s implementation of Seidel’s algorithm [[16]]. The results given in Figure 7 shows our algorithm to average about 40 times faster, for polygon sizes up to 1800 vertices.

Relative timings were also performed on random polygons (up to 550 vertices) generated by RPG [[1]]. Here our algorithm is in the order of an average of 14 times faster. It should be noted that these polygons do not have the characteristics found in a practical dataset.

VIII. Conclusion

A fast, practical, deterministic algorithm to decompose an arbitrary simple polygon into trapezoids has been presented. The polygon vertex coordinates can be specified as real numbers, and horizontal as well as vertical edges—very common in practical polygons—are explicitly handled. An intermediate output of the algorithm is a decomposition of the input polygon into a minimal set of monotone polygons. The fact that the trapezoids are output in useful groupings (trapezoid sequences), allows a further compression of polygon trapezoidal storage. A very fast linear scan of vertices is used to determine a collection of key vertices (joins), and these are processed in sweep order. No complicated data structures are needed. The practical complexity appear to be very close to linear. Nested holes were implemented as a simple extension.

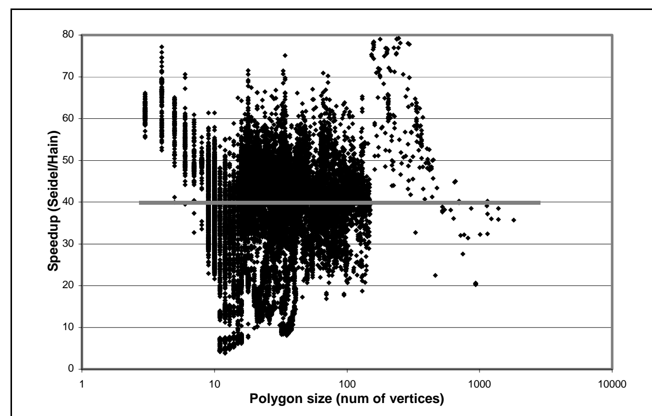

Figure 7 Relative execution times, Hain vs Seidel.

IX. Acknowledgements

We would like to thank QMS, Inc. for supporting this research. We would also like to thank Somnath Gulve for his help in building a dynamic testing tool to experiment with, and exhaustively test [[9]], the implementation of the algorithm.

References

[1] Auer, T., and Held, M. RPG: Heuristic for the generation of random polygons. Proc. 8th Canada Conf. Comput. Geom. Ottawa, Canada, Aug. 1996, 38–44.

[2] Chazelle B. A theorem on polygon cutting with applications. Proceedings of 23rd IEEE Symposium on Foundation of Computational Science, 339–349, 1982.

[3] Chazelle B. Triangulating a simple polygon in linear time. 31st IEEE Symposium on Foundation of Computational Science (1990), 220–230.

[4] Chazelle B., and Guibas L.J. Visibility and intersection problems in plane geometry. Discrete and Computational Geometry 4 (1989), 551–581.

[5] Chazelle B., and Incerpi J. Triangulation and shape-complexity. ACM TOG, 3(2):135–152, 1984.

[6] Clarkson K., Tarjan R.E., and Van Wyk C.J. A fast Las Vegas algorithm for triangulating a simple polygon. Discrete Computational Geometry, 4:423–432, 1989.

[7] Fournier A., and Montuno D.Y. Triangulating simple polygons and equivalent problems. ACM Transactions on Graphics, 3(2):153–174 , 1984.

[8] Garey M.R., Johnson D.S., Preparata, F.P., and Tarjan R.E. Triangulating a simple polygon. Information Processing Letter, 7:175–179, 1978.

[9] Hain T.F., Gulve S. Interactive, Visual Testing Strategy for Computational Geometry Problems. Proceedings of 36th Annual ACM Southeast Conference, Atlanta, Georgia, April 1–3, 1998

[10] Hertel S., and Melhorn K. Fast triangulation of simple polygons. Proc. of 4th International Conf. on Foundational Computational Theory, v.158 of Lecture Notes in Computer Science, 207–218. Springer-Verlag, 1983.

[11] Kirkpatrick D.G., Klawe M.M., and Tarjan R.E. Polygon Triangulation in $O(n \log \log n)$ time with simple data structures. Proceedings of 6th Annual ACM Symposium on Computation Geometry, 34–43, 1990.

[12] Laszlo M.J. Computational Geometry and Computer Graphics in C++, Prentice Hall, Englewood Cliffs, NJ, 1996.

[13] Lee D.T. and Preparata F.P. Location of a point in a planar subdivision and its application. SIAM Journal of Computation, 6:594-606, 1977.

[14] Lennes N.J. Theorems on simple polygon and polyhedron. American J. of Mathematics. 33:37–62, 1911.

[15] Mulmuley K. Computational Geometry: An Introduction Through Randomized Algorithms. Prentice Hall, Englewood Cliffs, NJ, 1994.

[16] Narkhede A., and Manochoa, D., Fast Polygon Triangulation Based on Seidel's Algorithm. A Technical Report, Dept. of Computer Science, UNC, Chapel Hill.

[17] O'Rourke J. Computational Geometry in C. Cambridge University Press, 1994.

[18] Preparata F.P., and Shamos M.L. Computational Geometry, An Introduction. Springer-Verlag, NY, 1985.

[19] Seidel R. A Simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. Computational Geometry Theory and Applications, 1:51–64, 1991

[20] Toussaint G. An output-complexity-sensitive polygon triangulation algorithm. Report SICS-86.3; McGill University, Montreal, 1988.

[21] Toussaint G. Triangulation and Arrangements. All Institute Lecture at McGill University, Montreal, 1993.