PERGAMON

Computers & Graphics 26 (2002) 281–289

COMPUTERS & GRAPHICS

www.elsevier.com/locate/cag

Technical Section

A fast trapezoidation technique for planar polygons

Gian Paolo Lorenzetto*, Amitava Datta, Richard C. Thomas

Department of Computer Science and Software Engineering, The University of Western Australia, 35 Stirling Highway, Nedlands, 6907 WA, Australia

Abstract

We present a simple and efficient algorithm for decomposing a planar polygon with holes into trapezoids. We show that our trapezoidation algorithm takes $O(n \log n)$ time, where n is the total number of vertices in the polygon and holes. The previous best algorithm for this problem by Žalik and Clapworthy (Computers and Graphics 23 (1999) 353) runs in $O(n^2 \log n)$ time. Our algorithm is extremely simple and we use only simple data structures. Our experimental results show that our algorithm is very fast in practice.

Keywords: Polygon decomposition; Trapezoidation; Computational geometry; Red–Black tree

1. Introduction

Planar polygon decomposition is the technique for decomposing complex geometric objects into simpler components. It has applications in many fields such as visibility computation, region filling problems and path planning. A common approach to the decomposition of planar polygons is *triangulation* [2]. Moreover, many triangulation algorithms use *trapezoidation* as a first step. A trapezoid is a four-sided polygon in which two of the edges are parallel. Trapezoidation was first considered by Chazelle and Incerpi [3] and Fournier and Montuno [4]. The first randomized algorithm for trapezoidation was given by Seidel [5]. The difference between triangulation and trapezoidation, other than differing shapes, is that the definition for trapezoidation allows for the creation of new vertices [3]. However in certain applications the number of primitives involved in trapezoidation is less and hence it is more efficient [1].

Of the algorithms mentioned above none consider polygons containing nested polygons, or holes. Žalik and Clapworthy [1] presented a universal algorithm to perform the trapezoidation of a non-monotone planar polygon containing holes. Their algorithm decomposes a simple [6], planar polygon containing holes into trapezoids, and requires $O(n^2 \log n)$ time.

We present a new approach to the trapezoidal decomposition of a planar polygon containing holes. It decomposes the same type of polygons as in [1] in $O(n \log n)$ time, an improvement by a factor of $O(n)$.

The algorithm presented is based on a scan line approach [7]. Fig. 1 illustrates a polygon decomposed into trapezoids. The scan lines used in the decomposition are drawn, and the first three generated trapezoids are marked T_1, T_2, and T_3. Note that a trapezoid may degenerate to a triangle, as for T_1.

The rest of the paper is organized as follows. In Section 2, we define terminology. Section 3 presents a short review of the Žalik and Clapworthy algorithm, and a brief analysis of its performance. The new algorithm is given in Section 4. We also discuss the comparison of the performance of the new algorithm with that due to Žalik and Clapworthy. Results of applying the new algorithm to various datasets are presented in Section 5, and Section 6 concludes with a summary of future research directions.

2. Terminology

We now give some notations and definitions before we discuss the algorithm by Žalik and Clapworthy.

*Corresponding author. Tel.: +61-8-9380-3778; fax: +61-8-9380-1089.

E-mail address: gian@cs.uwa.edu.au (G.P. Lorenzetto).

0097-8493/02/$ - see front matter
PII: S0097-8493(01)00180-7

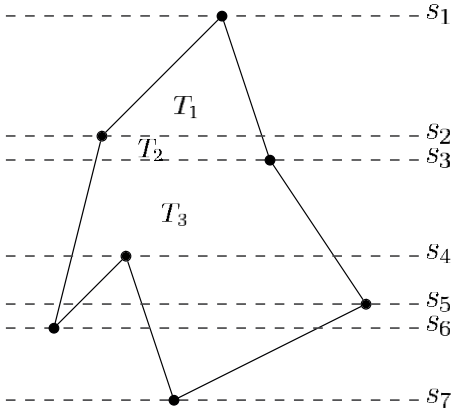

Fig. 1. A polygon decomposed into trapezoids via a scan line algorithm. Trapezoids are defined between the dotted lines. The first three trapezoids constructed are marked T_1, T_2, and T_3, respectively. The lines $s_1, s_2, \ldots, s_7$, represent scan lines used in the decomposition of the polygon. Each scan line passes through at least one polygon vertex.

Throughout this paper, we use the terms *loop* in reference to the outer, or enclosing polygon, and *hole* to indicate a polygon within the loop. Holes may be nested to any level.

A polygon is described by a collection of edges. Each edge is defined by a pair of polygon vertices. A polygon is said to be simple if its boundary does not intersect itself [6]. Thus a simple polygon has the restrictions that non-adjacent edges do not intersect, and adjacent edges only intersect at their common end point.

Both the Žalik and Clapworthy [1] algorithm and the presented algorithm are capable of decomposing simple polygons. Moreover, the given algorithms will also decompose polygons containing holes, and allow the sharing of both vertices and edges. This definition includes all simple polygons and a subset of non-simple polygons. Fig. 2 illustrates a polygon suitable for decomposition by both algorithms.

Under the above definition a coincident edge can exist between a hole and the loop, an unlikely scenario in GIS applications, but one applicable to other areas. In particular, the ability to trapezoid holes coincident with the loop is of value to the authors' future work in visibility, and 3D trapezoidation (see Section 6).

3. The Žalik and Clapworthy algorithm

The earliest trapezoidation algorithms, such as those proposed by Chazelle and Incerpi [3], and Fournier and Montuno [4], deal well with the trapezoidation of general polygons. The Fournier and Montuno [4] method also copes with holes, in that it is able to trapezoid only the inside of the loop (ignoring the holes

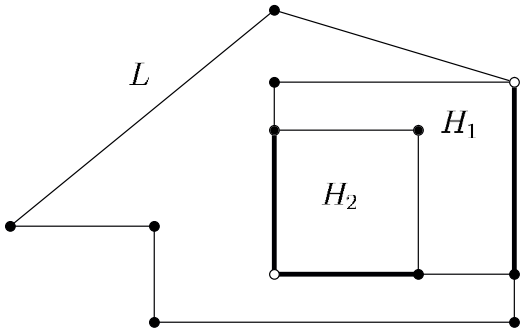

Fig. 2. A *valid* polygon. L is the loop, H_1 and H_2 are holes. White vertices are shared, as between holes H_1 and H_2, and between hole H_1 and the loop. Coincident edges are in bold. Holes H_1 and H_2 have two coincident edges, while the loop has a coincident edge with hole H_1.

all together). However the Žalik and Clapworthy [1] algorithm is the first trapezoidation technique presented that is able to trapezoid both the loop *and* holes. The algorithm performs three distinct phases of processing, and runs in $O(n^2 \log n)$ time, where n is the total number of vertices in the loop and holes. Moreover, their algorithm copes with shared vertices and shared edges.

We now give a brief review of the algorithm by Žalik and Clapworthy [1]. The algorithm uses a scan line approach, as depicted in Fig. 1, and is based on a simple analysis of each polygon vertex as given by O'Rourke [7]. Unlike O'Rourke's method, Žalik and Clapworthy's algorithm includes the possibility of multiple vertices on a scan line, so the range of vertex types is larger. The type of a vertex v_i is determined by its relationship with its neighboring vertices v_{i-1} and v_{i+1}. This classification is used to determine how a vertex is used in the generation of trapezoids. Types used to classify vertices are listed below and illustrated in Fig. 3.

- *Type INT*: Suppose the neighboring vertices of v_i have a greater and a lower y-coordinate, respectively, than v_i. The vertex v_i is then handled as if it were an intersection of the scan line with a polygon edge, and so it is categorized as *INT*.
- *Type MIN*: when both neighboring vertices have greater y-coordinates than v_i, then v_i is categorized as *MIN*.
- *Type MAX*: when both neighboring vertices have lower y-coordinates than v_i, P_i categorizes as *MAX*.
- *Types HMAX & HMIN*: when one of the neighbors of v_i has the same y-coordinate, then v_i is an end point of a horizontal line segment; by considering the position of the second neighbor, vertex v_i is given the type.

  - *Type HMAX* if the second neighbor has a lower y-coordinate or

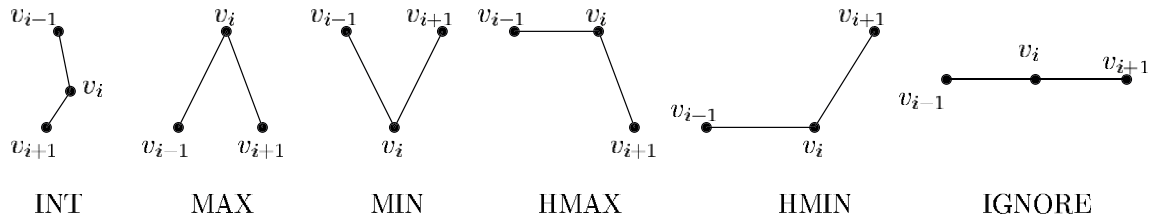

Fig. 3. Vertex types used in the Žalik and Clapworthy algorithm [1]. Note that the IGNORE type is not explicitly used.

- *Type HMIN* if the second neighbor has a higher y-coordinate.

The algorithm has three distinct stages: (i) initialization, (ii) decomposition, and (iii) post-processing.

Initialization: Initialization of the algorithm proceeds as follows:

- determine the type of each vertex,
- sort the vertices by y-coordinate, and
- sort the vertices on the first scan line into x-increasing order.

The total time requirement for this initialization is $O(n \log n)$.

Decomposition: Decomposition occurs in four steps, applied to each scan line in turn, as follows:

(1) determine the number of vertices lying on the current scan line,
(2) calculate intersections between the scan line and polygon sides,
(3) sort the points on the scan line into x-increasing order, including vertices and intersections, and
(4) create trapezoids between the current and previous scan lines.

Decomposition is completed in $O(n^2 \log n)$ time. The most costly operation in terms of time is the sorting of intersections, for each and every scan line.

If a hole is added to the polygon, the algorithm behaves similarly until the scan line passes through the hole. In this case, the hole is treated effectively as a new polygon, and a separate list of vertices and intersections is kept. If a hole is not to be decomposed, these additional lists can be discarded.

To avoid confusion, a *shared vertex* is treated as two separate vertices, one in each of the polygons sharing the vertex. That is, the shared vertex is treated as a set of coincident vertices.

Post-processing: Finally, a post processing stage is required to remove unnecessary trapezoids. Trapezoids are presented to the post-processing stage in the order that they are reported, sorted in order of x- and y-coordinates. Each trapezoid is tested against those above and below it to see if they can be joined. This process is completed in $O(n^2)$ time.

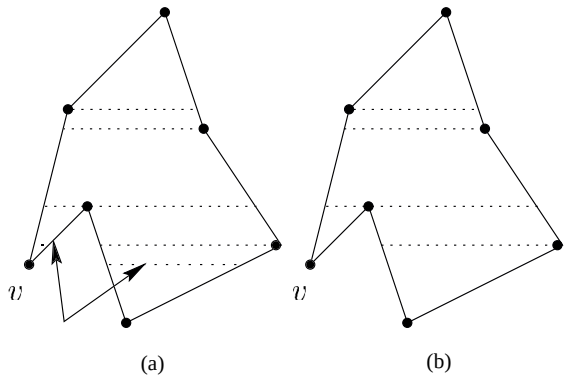

Fig. 4. Results of applying the Žalik and Clapworthy algorithm to the polygon (a) before the post-processing stage, and (b) after the post-processing stage. Note the extra trapezoids reported in (a), as indicated by the arrows.

The reason for a post-processing stage is that the algorithm calculates *all* intersections of each scan line with the polygon, producing many unnecessary trapezoids. Fig. 4(a) highlights the need for this. The algorithm generates more trapezoids than necessary due to the vertex v causing *all* intersections of the scan line through v and the polygon to be calculated. In fact, *none* of the intersections of the scan line passing through v are necessary. After the post-processing stage the polygon is correctly decomposed, Fig. 4(b).

4. Improved trapezoidation algorithm

We now present a new approach to the trapezoidal decomposition of a planar polygon that is based on a scan line method, like the Žalik and Clapworthy algorithm. However, the new algorithm is significantly different, consisting of only two distinct stages: a *pre-processing stage*, and the *main vertex processing stage*.

The *pre-processing stage* is used to significantly reduce the overall complexity of the algorithm. It also removes the need for a costly post-processing stage, because we set a flag for each vertex indicating whether a *sweepline*

should be thrown from a vertex, and if so in which direction.

Throughout the rest of the paper:

- the term *scan line* will be used to describe a horizontal line passing through at least one vertex, and
- the term *sweepline* will be used to indicate that a horizontal line has been thrown from a vertex, and its intersection with one, and only one, polygon or hole edge calculated.

The notion of a sweepline has been introduced because not all scan lines will be used in the generation of trapezoids. A major drawback of the Žalik and Clapworthy algorithm is that it generates unnecessary trapezoids. By distinguishing between scan lines and sweeplines, we are able to generate only the required trapezoids. Also, we do not calculate any unnecessary intersections of the scan line and polygon edges.

After pre-processing is complete, the second stage simply calculates all required intersections of the scan line with polygon edges. Trapezoids are then generated.

4.1. Pre-processing the polygon

The pre-processing stage is used to accomplish two goals.

1. We first calculate in which directions (if any) sweeplines should be thrown from a vertex. This can be determined by checking a vertex and its two neighboring vertices along the polygon boundary. Depending on the vertex types in Fig. 3, we can decide whether to throw a sweepline from that vertex and if so, in which direction. Either a clockwise or an anti-clockwise traversal of the polygon boundary is sufficient for this determination. This computation takes $O(n)$ time. A similar computation is done for each of the holes.
2. Next, we sort the vertices according to their x- and y-coordinates in ascending order. This determines the order in which the vertices have to be visited by the scan line in the vertex processing stage. This computation takes $O(n \log n)$ time. Note that, we sort all the vertices (including polygon and hole vertices) and create a list.

In our implementation the vertices are inserted into a Red–Black tree [8]. A Red–Black tree is a form of a binary tree that performs insertion and deletion operations in $O(\log n)$ time, where n is the number of elements in the tree. The ordering is by y- and then x-coordinate, and thus we can process the vertices in scan line order by processing the external nodes of the Red–Black tree.

Fig. 5 shows the result of applying the pre-processing step to a polygon. The subscripts indicate the order in which the vertices are stored in the tree, the arrows indicate the directions of the sweeplines.

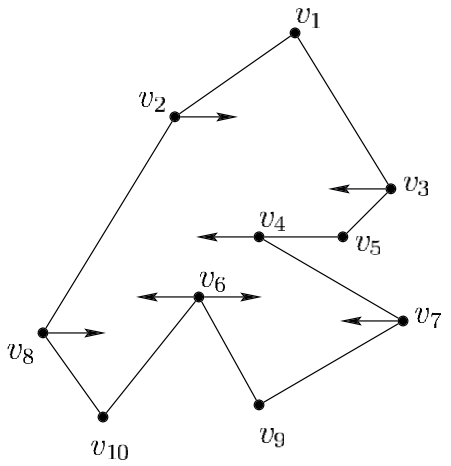

Fig. 5. After application of the pre-processing stage, each vertex of the polygon stores the directions sweeplines should be thrown (if at all). The vertices are numbered in scan line order.

4.2. Main vertex processing

The vertex processing stage of our algorithm makes use of an *edgelist*. The edgelist is another Red–Black tree structure, used to hold edges intersecting the current scan line. Each edge in the edgelist is stored as a pair of vertices. At every position of the sweepline, we execute the following five steps. Recall that the sweepline stops at every vertex.

Step 1: Adding or removing edges from the edgelist by applying both the following two rules to each vertex:

- If the vertex is the start of a previously unseen edge, add the edge to the edgelist.
- If the vertex is the end of a previously seen edge, mark it for removal from the edgelist.

Step 2: Look-up which directions sweeplines are to be thrown. Note that these directions have been already determined in the first stage for each vertex.

Step 3: If sweeplines are to be thrown, use the edge list to determine the polygon edges intersecting each sweepline.

Step 4: If sweeplines have been thrown, calculate necessary intersections, and generate trapezoids.

Step 5: Remove marked edges.

Note that in Step 1 both cases may be true: the vertex may be the end of one edge and the start of another, as for an INT vertex type. Table 1 describes what edge operations should be performed as we encounter particular vertex types. Since every vertex has two incident edges, we classify these two edges in the following way.

Table 1
Edge operations performed on the edgelist for various vertex types

Vertex type	Edge operation
MAX	Insert two edges
MIN	Remove two edges
HMAX	Add non-horizontal edge
HMIN	Remove non-horizontal edge
INT	Add next edge, and Remove previous edge

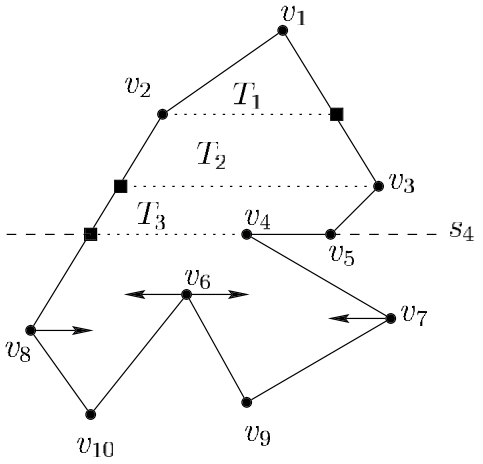

Fig. 6. Result of processing the first five vertices. Each black square indicates the intersection of a sweepline and a polygon edge. Three trapezoids have been reported, labeled T_1–T_3. s_4 indicates the position of the just completed sweepline.

Previous edge: We define the edge *previous* to a vertex v to be the edge that has v as an endpoint and that is inserted into the edgelist first.

Next edge: The *next* edge from v is the reverse, the edge that is inserted second into the edgelist.

Note that we do not ever add a horizontal edge to the edgelist, as it would be added and removed on the same scan line.

Fig. 6 depicts the polygon from Fig. 5 after processing the first five vertices. Table 2 lists the edges involved when processing scan line s_4, as well as the operations performed on each edge. At this time there is only one edge to be inserted, edge (v_4, v_7), as the horizontal edge (v_4, v_5) is not added, and the edge (v_2, v_8) was inserted on a prior scan line. Three trapezoids have been created, labeled T_1, T_2, and T_3. Each black square marks an intersection of a sweepline and polygon edge.

4.3. Operations on the edgelist

Our algorithm does not calculate all intersections of the current scan line with the polygon as in [1]. This is because the edgelist keeps an ordered list of all edges intersecting the current scan line and only calculates the intersections required to form trapezoids. In other words, if we throw a sweepline from the current vertex, we only determine the edge(s) which are first hit by the sweepline. There can be at most two edges if the sweepline has to be thrown in both directions, or one edge if the sweepline has to be thrown only in one direction. These edges which the sweepline first hits can be found by searching the Red–Black tree with the x-coordinate of the current vertex (from where the sweepline is thrown). This search takes $O(\log n)$ time for each position of the sweepline. We can also compute the intersection of the sweepline with the edge(s) it hits first. This takes $O(1)$ time once the edge or edges are found.

Table 2
Edges involved in processing scan line s_4 from Fig. 6. The operation performed on each edge is also listed

Edge	Operation
(v_2, v_8)	No change—remains in edgelist
(v_3, v_5)	Removed from edgelist
(v_4, v_5)	Ignored
(v_4, v_7)	Added to edgelist

Moreover, as each required intersection is calculated, the edge the intersection occurred on is updated such that its end point now coincides with the intersection point. This is done to simplify the reporting of trapezoids. Note that it is important to differentiate between intersections from the left and intersections from the right (see Fig. 8) for holes.

Depending on the type of the current vertex, the next task is to update the edge list either by inserting or by deleting edges. This is done as follows.

- If the current vertex is of type MAX (see Fig. 3) we have to insert two new edges in the edge list.
- If the current vertex is of type MIN, we have to delete two existing edges from the edge list.
- If the current vertex is of type HMAX, the non-horizontal edge is inserted, and the horizontal edge is ignored.
- If the current vertex is of type HMIN, the non-horizontal edge is removed, and the horizontal edge is ignored.
- If the current vertex is of type INT, one edge is inserted, and one edge is removed.

These operations are summarized in Table 1.

The deletion of an edge is straightforward. First, we search in the Red–Black tree with the end point of the edge. Once found it is removed. The balancing conditions of the Red–Black tree ensure that the tree will be

balanced after the deletion. However, insertion of an edge is more difficult as an edge must be inserted in the correct position in the edge list.

Insertion of an edge is performed as follows. Consider an edge e_i and its endpoint with the greater y-coordinate v_i, and call that our point of interest p. We compute a point p_∞, a point with very large x-coordinate on the current sweepline. This point can be determined easily if we know the largest x-coordinate among all the vertices in the polygon. The largest x-coordinate can be determined in $O(n)$ time as a part of the pre-processing stage. To ensure edges are inserted into the edge list in the correct position, it is necessary to perform a simple point-and-line test. The test is performed in the following way.

Point-and-line test: A binary search is performed on the edgelist to find an edge e_j such that p and p_∞ lie on the same side of e_j and the intersection of e_j with the current sweepline has the largest x-coordinate among all the edges in the edge list. The new edge e_i should be inserted in between e_j and e_{j+1} (the edge next to e_j) in the edge list. Since it takes $O(1)$ time to check whether two points are on the same side or on opposite sides of a line, this binary search can be completed in $O(\log n)$ time.

Fig. 7 shows the state of the edge list during the point-and-line test. The start vertex of the edge to be added e_i is the point p. Edge e_2 is the edge with the largest x-coordinate such that both p and p_∞ lie on the same side of the edge. Thus the edge with start vertex at point p will be inserted between edges e_2 and e_3.

The edge list is useful in calculating intersections because by using the point-and-line test described above, it provides a simple and fast way of determining the intersecting edge. Consider the case where we have just reached a particular vertex on a scan line and it indicates that we should throw a sweepline left. We use the vertex as our point p, and assume some p_∞ has already been calculated. The edge we are looking for must be the edge immediately to the left of p. Similarly, we look for the edge immediately to the right of p.

Suppose the vertex v_i is the end point of a previously unseen edge. Consider v_i as the point of interest p.

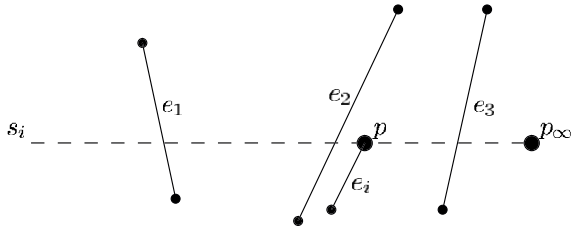

Fig. 7. Edges intersecting the current sweepline s_i. p is our point of interest, and p_∞ a point on the sweepline outside the polygon.

Clearly the new edge must be inserted, and this operation is equivalent to locating the point p in the list. If the point p is the end point of an edge already seen, then it must have already been inserted into the edge list. In both cases, the location of p in the edge list is known.

Once the intersections have been calculated, the new trapezoid can be reported in $O(1)$ time in the following way. We assume for simplicity that two intersections of the sweepline with two edges have been found. The other case is similar. Note that we record with each edge its intersection with the most recent sweepline. When the new intersection is found, the part of the edge between the old intersection point and the new intersection point becomes one edge of the trapezoid to be reported. Similarly the other edge of the trapezoid is determined. These two edges form the two non-parallel sides of the trapezoid. The parallel sides of the trapezoid are simply the portions of the previous and the current sweepline which fall between these two non-parallel edges.

The first step of the vertex processing stage is the most time consuming, as insertion into the Red–Black tree takes $O(n \log n)$ time. The sweep line direction look-up and the determination of the intersecting edge take $O(\log n)$ time in total. Since there are n vertices where the sweep line stops, the total time required for this stage of the algorithm is $O(n \log n)$. Hence the overall time requirement for stages 1 and 2 is $O(n \log n)$.

The decomposition process is now complete. We do not require any post processing stage unlike the Žalik and Clapworthy algorithm. The total time complexity of the new algorithm is $O(n \log n)$. Hence, the new algorithm improves the algorithm by Žalik and Clapworthy by a factor of $O(n)$.

4.4. Holes

It is generally easy to deal with holes, as most hole vertices will throw a sweepline in both directions. A hole may have the loop as its parent, or another hole and can be nested to any level. Since a hole is inside the loop, there is no chance we will throw the sweepline outside the polygon, except in the case of shared vertices or coincident edges, as discussed in Section 4.5. It is also not necessary to throw the sweepline in both directions if the current vertex lies on a horizontal line segment. We only need to use the vertices at the extreme left and right of a horizontal line segment to calculate the left and right intersections, respectively. This procedure is summarized as follows. For each hole vertex,

- if the vertex is not part of a horizontal edge, throw sweeplines to both the left and right,
- if the vertex is a HMIN or HMAX, throw sweepline in the direction opposite the horizontal segment, and

- if the vertex is of type IGNORE, ignore the vertex completely.

This process is illustrated in Fig. 8.

4.5. Shared vertices and coincident edges

Extending the algorithm to deal with holes introduces the possibility that a hole may share a vertex with the loop or another hole. Moreover, a hole may share an edge with the loop or another hole. This latter case is referred to as coincident edges. We now explain how to deal with both cases.

Recall that each hole vertex must be inside the loop, shared with a loop vertex, or lying on the loop in the case of a coincident edge. In the general case of a hole vertex lying inside the loop, we can safely throw sweeplines in both directions, as we are assured of throwing a sweepline inside the loop. Consider Fig. 8 and in particular the shared vertex v_{10}. In this case the two holes H_1 and H_2 are *sharing* a vertex v_{10}, and that the type of vertex v_{10} in H_1 is different from the type of v_{10} in H_2. The vertex v_{10} in H_1 is of type INT, and in H_2 of type HMIN, as illustrated in Fig. 9. Although we need to throw a sweepline to the left from v_{10} in H_1, we do not need to throw a sweepline to the left from v_{10} in H_2. Given a set of shared vertices we throw a sweepline in a particular direction only if *all* vertices throw a sweep line in that same direction. This is because:

1. the hole may be sharing a vertex with the loop, and we do not want to throw a sweepline outside the loop; and
2. the hole may be sharing a vertex lying on a horizontal line segment, along which we do not need to throw a sweepline.

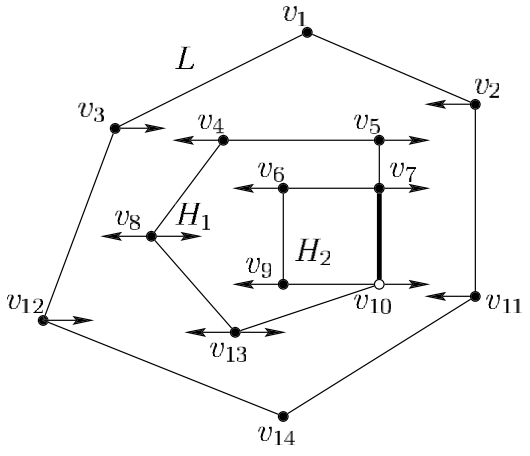

Fig. 8. Sweepline directions calculated for each vertex of the polygon. In this instance each hole vertex sweeps both left and right, except in the case of a horizontal line segment. Hole H_2 is nested within H_1. Edge (v_7, v_{10}) is coincident with edge (v_5, v_{10}). H_1 and H_2 also share a vertex v_{10}. The vertex subscripts indicate the order in which vertices are visited.

In the case of two holes sharing a vertex we need only check for horizontal line segments, as illustrated in Fig. 8.

When dealing with shared vertices it is important that we account for *all* shared vertices before deciding on which directions, if any, sweeplines should be thrown. Recall that we examine vertices in scan line order. Consequently, we will see all shared vertices one after the other. During the main vertex processing stage it is enough to use only one of the set of shared vertices. Unfortunately, we must now be careful when reporting trapezoids, because the type of vertex and directions sweeplines are thrown no longer match. Consider two shared vertices v_h and v_l, v_h belongs to a hole and v_l belongs to the loop. Let v_h be of type MAX. Thus sweeplines would normally be thrown both left and right from v_h. In this case, however, the hole vertex must *not* sweep outside the loop. Consequently, we must take this into account when generating trapezoids.

We must also consider that a shared vertex may actually be part of a coincident edge. If two edges are coincident, the criteria for inserting an edge into the edgelist is as follows. Let e be the current edge coincident with some edge in the edgelist.

- if e is longer than the edge already in the edgelist, remove the old edge and insert the new,
- otherwise, ignore the edge.

Note that all previous intersections with the edge to be replaced must be preserved. This amounts to simply moving the end point of the edge currently in the edgelist to the end point of e. Also, although the edge may be ignored, sweeplines may still be thrown from the vertices of the coincident edge, and trapezoids generated.

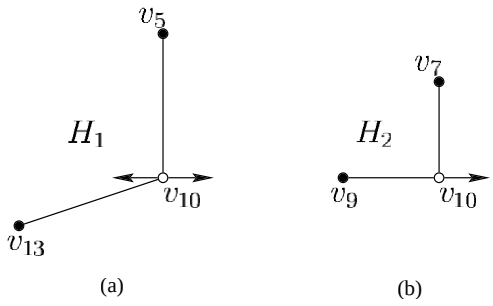

Fig. 9. Example of a vertex v_{10} shared between two holes, taken from Fig. 8 and highlighting the different classification of v_{10} in the different polygons. (a) The shared vertex type and sweep direction in hole H_1. (b) The shared vertex type and sweep direction in hole H_2.

The tests for shared vertices and coincident edges can be combined as follows. As we encounter a shared vertex v,

- if v is part of a coincident edge, proceed as for a coincident edge,
- otherwise, proceed as for a shared vertex.

Fig. 10 is an example of a polygon containing nested holes. Edge (v_7, v_{10}) from hole H_2 is coincident with edge (v_5, v_{10}) from hole H_1. The fifth sweepline s_5 is marked. Fig. 11 depicts the state of the edgelist at the completion of s_5. To ensure the edgelist is consistent, edge (v_7, v_{10}) must not be added. Rather, the edge already inserted into the edgelist (v_5, v_{10}) is updated as described above. In this case, the coincident edge to be inserted is not longer than the edge already inserted, thus the edge can be ignored.

Trapezoids must still be generated from vertex v_7 to correctly decompose the polygon.

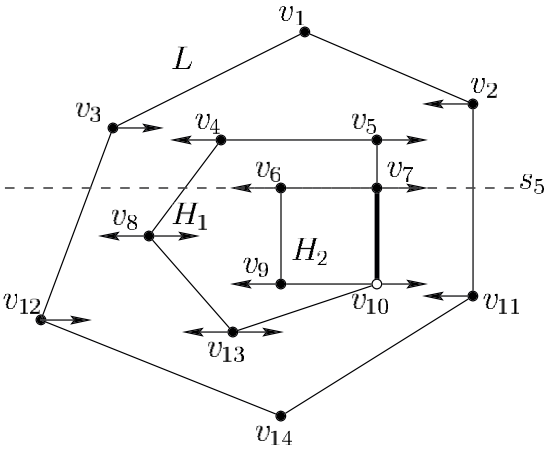

Fig. 10. The polygon from Fig. 8, with the fifth sweepline used in the decomposition marked.

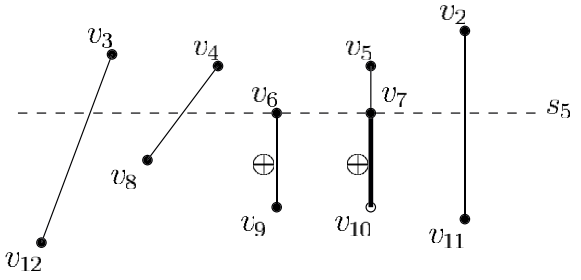

Fig. 11. The state of the edgelist at the completion of the fifth sweepline from Fig. 10. Edges marked with a $\oplus$ are newly added. The coincident edge (v_7, v_{10}) is not actually added. Rather, the edge already inserted into the edgelist (v_5, v_{10}), coincident with edge (v_7, v_{10}), is updated according to Section 4.5.

The algorithm allows a hole to have an edge coincident with an edge of the loop. In the above example the coincident edge was between two holes. If the coincident edge was between the loop and a hole, then a small check is added to modify the vertex such that it does not sweep outside the loop. However, if a hole does have an edge coincident with a loop edge, it cannot be longer than the loop edge, otherwise it would be outside the loop. Hence it is possible in this case to ignore completely the hole edge and use the already inserted loop edge.

It cannot be assumed that a coincident edge between two holes is longer than another, as shown in Fig. 2. The coincident edges *overlap*, hence the edges must be updated following the process outlined above.

5. Results

The following results were obtained from running the presented algorithm over various data sets. The program was implemented in C under Linux, and all tests were run on a Celeron 366 with 64MB of main memory. Both artificially constructed and actual GIS data have been used in testing the program. GIS data was freely obtained from the GIS Data Depot, at `http://www.gisdatadepot.com`. Fig. 12 is a representative data set with several thousand vertices.

Timing information was generated from averaging multiple runs of the program. A single run consisted of executing the program multiple times over the same set of data and averaging the resulting time.

Table 3 shows the results of running the application over a series of artificially created monotone polygons. Note the almost linear increase in time versus number of vertices.

Applying the program to polygons with many concave parts returned the results in Table 4. Note the

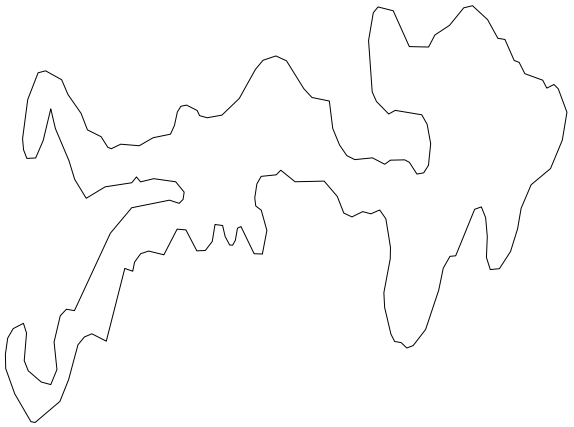

Fig. 12. A typical GIS data set.

Table 3
Simple, monotone polygon

Number of vertices	1000	1500	2000
Number of trapezoids	750	1125	1500
Total time (ms)	0.021	0.033	0.046

Table 4
Concave polygon, containing holes and featuring many horizontal lines

Number of vertices	1200	2400
Number of holes	1	2
Number of trapezoids	341	629
Total time (ms)	0.02	0.039

Table 5
Results for actual GIS data

Number of vertices	1000	1500	2000
Number of holes	71	132	173
Number of trapezoids	1965	2963	3961
Total time (ms)	0.029	0.0465	0.062

Table 6
Polygon containing many concave parts, and holes, using Žalik and Clapworthy [1]

Number of vertices	100	150	200
Number of holes	0	5	10
Number of trapezoids	92	175	262
Total time (ms)	16.1	32.9	45.9

small number of trapezoids generated for the two polygons. The data used for these tests contained many horizontal line segments. This is the ideal data set for a scan line based approach, as many vertices can be ignored, and thus do not cause trapezoids to be generated. For a more detailed discussion of this issue see Žalik and Clapworthy [1].

Results for actual GIS data are in Table 5. These data sets contain many concave parts, and holes, hence the proportion of trapezoids to vertices is high. As expected, in practice the algorithm performs very well with results approaching linear time.

Žalik and Clapworthy state the results in Table 6 for data similar to that in Table 5. Comparatively, increasing the number of vertices has a much greater impact on the Žalik and Clapworthy algorithm. As an example, consider the running times in Table 6 and in Table 5. The running time of the algorithm by Žalik and Clapworthy increases by almost a factor of 3 when the number of vertices is doubled. On the other hand, in our algorithm the running time increases almost linearly with increasing number of vertices. These results reflect the $O(n \log n)$ complexity of our algorithm as compared to the $O(n^2 \log n)$ complexity of the algorithm by Žalik and Clapworthy.

6. Further work and conclusion

We have presented a new algorithm for the decomposition of valid, planar polygons into trapezoids. To the authors' knowledge the Žalik and Clapworthy algorithm [1] is the only other trapezoidation algorithm capable of decomposing the main loop, and holes, of a set of polygons. The new algorithm improves significantly on the time performance of the Žalik and Clapworthy method. We have implemented the algorithm and the running times of our algorithm for large polygons show that the algorithm is very fast in practice.

The authors have a particular interest in visibility computation, and the application of decomposition routines in this area. Although not immediately obvious 2D decomposition routines can be applied to certain 3D scenes, such as 3D game environments, and any scene that can be reduced to a $2\,1/2$D representation.

Further work in this area will involve the extension of the decomposition algorithm to $2\,1/2$D, and eventually to 3D environments. This will be in conjunction with visibility and occlusion issues, and the potential reuse of pre-rendered cells.

References

[1] Žalik B, Clapworthy GJ. A universal trapezoidation algorithm for planar polygons. Computers & Graphics 1999;23:353–63.
[2] Bern M. Handbook of discrete and computational geometry. Boca Raton, FL: CRC Press, 1997 [chapter 22] p. 413–28.
[3] Chazelle B, Incerpi J. Triangulation and shape complexity. ACM Transactions Graphics 1984;3(2):135–52.
[4] Fournier A, Montuno DY. Triangulating simple polygons and equivalent problems. ACM Transactions Graphics 1984;3(2):153–74.
[5] Seidel R. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. Computational Geometry: Theory and Applications 1991;1(1):51–64.
[6] Mulmuley K. Computational geometry: an introduction through randomized algorithms. Englewood Cliffs, NJ: Prentice-Hall, 1994.
[7] O'Rourke J. Computational geometry in C. Cambridge: Cambridge University Press, 1993.
[8] Wood D. Data structures, algorithms, and performance. Reading, MA: Addison-Wesley, 1993.