

Triangulating Simple Polygons and Equivalent Problems

ALAIN FOURNIER and DELFIN Y. MONTUNO

University of Toronto

It has long been known that the complexity of triangulation of simple polygons having an upper bound of $\mathcal{O}(n \log n)$ but a lower bound higher than $\Omega(n)$ has not been proved yet. We propose here an easily implemented route to the triangulation of simple polygons through the *trapezoidization* of simple polygons, which is currently done in $\mathcal{O}(n \log n)$. Then the trapezoidized polygons are triangulated in $\mathcal{O}(n)$ time. Both of those steps can be performed on polygons with holes with the same complexity.

We also show in this paper that a number of problems, such as the decomposition of simple polygons into convex, star, monotone, spiral, and trapezoidal polygons and the determination of edge-vertex visibility, are linearly equivalent to the triangulation problem and therefore share the same lower bound. It is hoped that this will simplify the task of reducing the gap between the lower and upper bound for these problems.

Categories and Subject Descriptors: F.2.2 [Theory of Computation]: Non-numerical Algorithms and Problems—*Geometric problems and computations*; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—*geometric algorithms, languages, and systems*

General Terms: Algorithms, Computational Geometry

Additional Key Words and Phrases: Decomposition, scan conversion, triangulation, trapezoid

1. Introduction

In computational geometry many useful algorithms have been developed for decomposing objects into simple elements. Most notable are the decomposition of simple polygons into convex parts (with Steiner points [3] or without Steiner points [13, 21]), the triangulation of a set of points [18, 23], the triangulation of planar regions [14], the triangulation of simple polygons [11, 12], the decomposition of simple polygons into star polygons [2], and the trapezoidization of simple polygons [15, 24].

The motivation for such algorithms, beyond their intrinsic theoretical interest, is that further processing is usually easier on the resulting figures, and it is hoped that

this will more than offset the increase in storage and processing time due to their larger number. In algorithms commonly found in computer graphics, for scan conversion and shading in particular, it is often essential, for speed and simplicity of hardware implementation, to manipulate only convex polygons [8, 9], trapezoids [7, 15, 24] or triangles [10]. Triangles, in particular, are helpful for two-dimensional function interpolation (Gouraud/Phong shading being a specific example), where the result should be independent of the orientation of the triangles.

In this paper, we describe an algorithm and its implementation for triangulating an arbitrary simple polygon with holes, using trapezoid decomposition as a first step. Although this algorithm has the same $\mathcal{O}(n \log n)$ complexity as the known triangulation algorithms [11, 12], it is substantially simpler to implement. More important, we show that one can go in linear time from trapezoid decomposition to a triangulation. We also show the converse, that one can go in linear time from triangles to trapezoids. It is also important to note that the Steiner vertices introduced by the trapezoidization need not be computed explicitly, and they are not included in the triangulation. Another algorithm shows that the triangulation of star polygons is an $\Theta(n)$ process. We then obtain the beginning of a class of *triangulation-linear* problems, such that each member of the class is linearly related to any other, and any lower bound greater than or equal to $\Omega(n)$ proved for one applies to all of them. We extend this class to include other problems, such as the decomposition of simple polygons into star, monotone, and spiral polygons and the determination of all vertex-edge visibility for a simple polygon.

2. Decomposition into Trapezoids

We use the terms *trapezoidize* and *trapezoidization* for convenience and brevity, though not for euphony. The usual definition of a *trapezoid* is that it is a four-sided polygon with two parallel edges. We constrain all the trapezoids of the decomposition to have their parallel edges parallel to each other as well. Without loss of generality we assume that these parallel edges are also parallel to the x axis (i.e., horizontal). Thus a trapezoid can be fully specified as two line segments parallel to the x axis, known in computer graphics as scan line segments, for obvious reasons. If one of these segments degenerates to a point, the trapezoid becomes a triangle, which we still consider a trapezoid. The usual problem of decomposing a simple polygon into trapezoids can be stated as follows:

To find a minimal set of disjoint trapezoids covering the polygon.

It is easy to show that such trapezoids will have segments of the polygon edges as their nonhorizontal edges, and their horizontal edges going through at least one of the polygon vertices. Therefore, if we scan the polygon vertices in y order (either from top to bottom or bottom to top), a trapezoid of the decomposition will start and end only at vertex coordinates.

2.1. Convex Polygons

The trapezoidization of convex polygons is straightforward and well-known in computer graphics as a *scan-conversion* algorithm. It consists of finding the maximum (for top-to-bottom scanning) y vertex, which splits the polygon into two lists, a right one and a left one, both monotone in decreasing y coordinates. The method is to simply scan down the two lists to find the next vertex, creating a horizontal edge for the beginning and ending trapezoid, until the bottommost vertex is found (see Figure 1). This is then a $\Theta(n)$ process.

2.2. Simple Polygons

The vertices of simple polygons can be classified into three types, depending on the position of their adjacent edges with respect to the horizontal lines going through the vertices. Type 1 (called *regular* in [15]) vertices are the vertices with adjacent edges on both sides of the horizontal line and mark the end of a trapezoid and the beginning of another. Type 2 (called *stalagmitic* in [15]) are the vertices with both adjacent edges below the horizontal line. When a type 2 vertex is met, it marks the end of one trapezoid and the beginning of two new ones if the neighborhood above the vertex is inside the polygon, or the beginning of a trapezoid otherwise. Type 3 vertices (called *stalactitic* in [15]) are vertices with both adjacent edges above the horizontal line. In this case two trapezoids terminate and a new one appears if the neighborhood below the vertex is inside the polygon, or a trapezoid terminates otherwise (see Fig. 2).

We can now sketch out the algorithm used to trapezoidize a simple polygon.

Algorithm 1: Decomposition of a Simple Polygon into Trapezoids

Input. A polygon $P(v_0:v_{n-1})$ with vertices given in clockwise order. Note that in the rest of the paper we assume that the collinear adjacent edges have been eliminated by a $\mathcal{O}(n)$ preprocessing and that there are no horizontal edges. Of course, in the actual implementation, horizontal edges are allowed and handled correctly (see Fig. 7).

Output. A trapezoidized polygon $TP(v_0:v_{n-1})$, where each v_i has pointers to the second vertex v_j , defining the trapezoid, and to the two edges e_{left} and e_{right} , defining the sides of the trapezoid. The edges are defined by the first vertex of the edge in the polygon list. Each trapezoid appears only once, with the vertex of the highest y coordinate. Note that some vertices (of type 2) may point to two trapezoids (in this case the first one is the leftmost), and some vertices (of type 3) may not point to any trapezoid.

Data Structures. In addition to the above, the algorithm uses a 2–3 tree holding the *active* trapezoids as leaves (a trapezoid is active if intersected by a horizontal line between the last vertex processed and the next one). The edges (right and left) are the keys used for comparison in searching. We use the fact that while a trapezoid is active, a vertex is inside the trapezoid if and only if it is inside its left edge and its right edge. This is true since we know the vertex being tested has to have a y coordinate within the closed interval defined by the coordinates of the two edge boundaries. This

avoids having to compute the intersection of the edges with the current y horizontal line. The insertion test in practice is done only for type 2 vertices. For the other types, at least one of their edges already appeared in the data structure, and can be found in constant time with a suitable pointer.

```

sort all vertices in decreasing order of y coordinates
    and increasing order of x coordinates
    for vertices of equal y coordinates;
for each vertex  $v_i$  in order do
begin
determine type of  $v_i$ ;
case:
    type 1: begin
        find edge in 2-3 tree;
        add  $v_i$  to complete trapezoid structure;
        remove trapezoid from 2-3 tree;
        replace edge to which  $v_i$  is adjacent
            by other adjacent edge of  $v_i$ ;
        insert new trapezoid structure with  $v_i$ 
            and the right and left edges;
        end
    type 2: begin
        search for location in 2-3 tree;
        if  $v_i$  is within an active trapezoid then
            begin
            add  $v_i$  to complete trapezoid;
            remove trapezoid from 2-3 tree;
            insert two new trapezoid structures
                with  $v_i$  and its edges in addition
                to the former trapezoid edges;
            end
        else begin
            insert new trapezoid with  $v_i$  and
            its two edges
        end
    type 3: begin
        find adjacent edges in 2-3 tree;
        if edges belong to same trapezoid then
            begin
            complete trapezoid by adding  $v_i$ ;
            remove trapezoid from 2-3 tree
            end
        else begin
            complete right and left trapezoids
                by adding  $v_i$  to their structures;
            remove both trapezoids from 2-3 tree;
        end
    end
end

```

```

    insert new trapezoid with  $v_i$  and the
        left trapezoid left edge and the right
        trapezoid right edge
    end
end case
end

```

Analysis. The running time of the above algorithm is $\mathcal{O}(n \log n)$. The initial sorting can be done in $\mathcal{O}(n \log n)$ time by a variety of algorithms. We used heapsort in practice, since the code is short and easy to read. The main loop is executed n times, and each of its steps has a constant execution time, except for searches into the 2–3 tree, which is $\mathcal{O}(\log k)$, where k is the number of leaves in the tree. Since $k = \mathcal{O}(n)$, the whole loop is $\mathcal{O}(n \log n)$ in the worst case. Note that the average case is $\mathcal{O}(n \log k)$, where k is the average number of edges intersecting the horizontal lines going through the vertices. This number is usually low (and this degenerates to $\mathcal{O}(n)$ for convex polygons). We have, therefore:

Theorem 2.1. *A simple polygon can be trapezoidized in $\mathcal{O}(n \log n)$ time.*

Beyond the asymptotic complexity, which is the same as in [15], the important characteristic of this algorithm is that it uses only simple and “safe” operations. It especially should be noted that the coordinates of the Steiner vertices in the trapezoids are never computed. The tests for search in the 2–3 tree are only to determine which side of a line a vertex is. This involves four subtractions, two multiplications, and a compare, and since the vertices defining the edge itself are not tested in this way, most of the difficulties involved with points very close to a line are eliminated. Vertices with the same y coordinates are handled correctly. In the initial sort they are sorted in x increasing order. It is only necessary to ensure that the inside–outside test gives consistent results when an edge is horizontal and the tested vertex has the same y coordinate. If two vertices have the same y coordinate and are not adjacent, then they are processed in their x coordinate order. Note that in both cases trapezoids of zero height can be created, and the following algorithms should handle them correctly.

2.3. Polygons with Holes

In the preceding algorithm, the connectivity of the boundary of the polygon was not explicitly used. Thus it is trivial to extend this algorithm to handle polygons with holes, to any level of recursion. If we define a hole in a polygon to be another polygon completely enclosed by it, and make sure that it is given in counterclockwise order if the surrounding polygon is given in clockwise order, we then can prove that the above algorithm will correctly decompose only the inside of the polygon into trapezoids. The change from clockwise to counterclockwise is necessary for the inside–outside test when inserting into the 2–3 tree.

If a hole has some of its edges involved in some of the active trapezoids during the execution of an algorithm, then the decomposition is correct so far, since at this point

it is not possible to distinguish the hole from the outside of the polygon. When the vertex of a minimum y coordinate is processed for a hole, it will be of type 3, and Algorithm 1 will execute the **else** branch of its case of “close” the hole correctly. The processing of the hole will then be terminated correctly.

By a similar reasoning, we can show that the algorithm will correctly handle “islands” within holes, and in fact any number of disjoint polygons, as long as they do not intersect and are described in consistent order. This leads to the following:

Theorem 2.2. *Simple polygons with holes which are simple polygons can be trapezoidized in $\mathcal{O}(n \log n)$.*

3. Triangulation from Trapezoids

We now describe the algorithm for triangulating the polygon from the trapezoids. It is based on the simple observation that the trapezoids can be put into two classes: *class A*, where the two vertices involved share an edge of the trapezoid (and therefore of the original polygon), and *class B*, where they do not. In the latter class, it is then possible to create one of the triangulation edges between the two vertices (see Figure 3). When this is done for all the trapezoids in this class, we are then left with subpolygons that are unimonotone with respect to the y axis. Here we define *unimonotone* polygons in a more restrictive sense than monotone polygons [11]. We call a polygon $(P_0, P_1, \dots, P_{n-1})$ unimonotone if there is an i such that P_i and P_{i+1} are the vertices with maximum and minimum y coordinates (either order) and $P_{i+2}, P_{i+3}, \dots, P_{i-1}$ are in nondecreasing or nonincreasing order of y coordinates (all indices are *modulo n*, see Figure 4).

These unimonotone polygons can be triangulated in linear time with the algorithm given below.

Algorithm 2: Triangulating the Trapezoidized Polygon

Input. Same as output of algorithm above. **prev**(vertex) and **next**(vertex) denote the predecessor and the successor of *vertex* in the polygon list.

Output. Same as above, with each vertex pointing to its list of adjacent triangles.

```

triangulate(first, last)
current_vertex = first;
while not current_vertex.done do
    begin
    current_vertex.done = TRUE;
    bottom_vertex = diagonal(current_vertex);
    if bottom_vertex not NULL then
        begin
        save_next = next(current_vertex);
        save_prev = prev(bottom_vertex);
        next(current_vertex) = bottom_vertex;

```

```

prev(bottom_vertex) = current_vertex;
trapezoid(current_vertex) = NULL;
triangulate(bottom_vertex, current_vertex);
current_vertex.done = FALSE;
bottom_vertex.done = FALSE;
next(current_vertex) = save_next;
prev(bottom_vertex) = save_prev;
next(bottom_vertex) = current_vertex;
prev(current_vertex) = bottom_vertex;
triangulate(current_vertex, bottom_vertex);
return
end
else current_vertex = next(current_vertex);
end
triangulate_monotone(first, last)

```

Analysis. The function **diagonal**(*vertex*) returns the bottom vertex of the trapezoid pointed at by *vertex* if they do not share an edge of the polygon. It removes the trapezoid from the vertex structure in this case. It returns NULL if they do, or if there is no trapezoid pointed at by *vertex*. The salient point of the algorithm is that, when the call to **triangulate_monotone** is reached, the polygon (*first*, *last*) is indeed unimonotone. To prove this, assume that this is not the case, and that there is at least one pair of vertices between *first* and *last* whose *y* coordinates are out of order. Then at least one vertex is of type 2 or 3, its trapezoid belongs to class B and for this vertex **diagonal()** would return a non-NUL pointer, contradicting the fact that the **triangulate_monotone** call has been reached (if *bottom_vertex* is not NULL, the algorithm returns from this invocation of **triangulate()**).

If the polygon contains holes, the only amendment to the preceding algorithm is to check, when a diagonal is found, whether it is between the polygon and a hole or between two holes. If it is, then the current and bottom vertices should be duplicated to introduce two edges, one in the current-bottom direction and the other in the bottom-current direction. This will in effect remove a hole from the polygon. Then we proceed as before.

We postpone the analysis of the running time of this algorithm until after we examine **triangulate_monotone**.

Algorithm 3: Triangulating Unimonotone Polygons

Input. A unimonotone polygon (with respect to the *y* axis), in the same format as polygons described above.

Output. The same polygon, with the list of adjacent vertices in triangles in place.

```

triangulate_monotone(first, last)
begin
determine start vertex (topmost if the monotone chain is on the right,
bottommost if it is on the left).

```

```

determine number_of_vertices;
current = next(start);
while number_of_vertices ≥ 3 do
begin
if angle(prev(current), current, next(current)) is CONVEX then
begin
for each of prev(current), current, next(current) do
begin
insert other two vertices to form triangle;
and
save = prev(current);
remove current from uni-monotone polygon;
if current = first then current = next(first)
else current = save;
decrement number_of_vertices
end
else current = next(current)
end
end

```

Analysis. The initializing steps take at most $\mathcal{O}(n)$ time. The unimonotone polygons, as defined above, have at least one vertex besides the topmost and bottommost ones whose internal angle is convex, since the sum of their $n - 2$ internal angles is between $(n - 3) \times \Pi$ and $(n - 2) \times \Pi$ (see Figure 5; we consider an angle less than or equal to Π convex). Therefore this vertex will be found in the first test inside the **while** loop. There cannot be other vertices of the polygon inside the triangle formed by the *current* vertex and its two adjacent vertices, since this would violate the unimonotone property of the polygon (the *y* coordinate of such a vertex would have to be between the *y* coordinates of **next**(*current*) and **prev**(*current*)). So a triangle of the triangulation can be formed and the *current* vertex removed from the polygon. This leaves a unimonotone polygon (since we do not remove the topmost or bottommost vertex), and, therefore, by induction we deduce that the algorithm will proceed until only three vertices are left, in which case the angle is necessarily convex, the last triangle will be recorded, and it will terminate. The algorithm backtracks by one vertex only when a vertex is removed, so there are at most $\mathcal{O}(n)$ backward steps and $\mathcal{O}(n)$ forward steps. We now can state the following theorem about unimonotone polygons:

Theorem 3.1. *Unimonotone polygons can be triangulated in $\Theta(n)$ steps.*

Since the previous triangulation algorithm breaks polygons into unimonotone polygons in $\mathcal{O}(n)$, we can now state:

Theorem 3.2. *Trapezoidized polygons can be triangulated in $\Theta(n)$ steps.*

Which allows us to state:

Theorem 3.3. *Simple polygons with holes (themselves simple polygons with holes) can be triangulated in $\mathcal{O}(n \log n)$.*

The statement of the theorem can include polygons with holes, since the first step (trapezoidization) handles them correctly, the second step (breaking into unimonotone polygons) is correct with holes with the modification indicated, and by the third step (triangulation of monotone polygons) holes have disappeared from the structures.

This does not improve upon the previously known upper bound, but, again, the main advantages of this method are that it is simple to implement; uses a basic algorithm, the scan conversion of polygons, familiar to practitioners in computer graphics; and uses only one arithmetic operation on the coordinates, the cross-product, for search and insertion into the 2–3 tree and for checking angle convexity. This operation involves four subtractions, two multiplications and a compare. The operations can be done on the data type of the input coordinates, without conversion, for instance in integer or fixed point arithmetic, which is generally faster and less troublesome than floating point. The only error conditions possible are overflows, and these can be easily avoided if the range of coordinates is known.

The $\mathcal{O}(n \log n)$ complexity can be improved to $\mathcal{O}(r \log r + n)$, where r is the number of type 2 vertices, by modifying the trapezoidization step (Algorithm 1) in the following way.

In the sorting step, sort only the type 2 vertices, in decreasing order of their y coordinates. This will be an $\mathcal{O}(r \log r)$ step. Each type 2 vertex determines the start of two monotone chains (monotone with respect to the y axis) which are ended by type 3 vertices. Within those monotone chains, the vertices do not need to be sorted, as they are in the order in which they appear in the polygon. In the **for** loop of Algorithm 1, the next vertex of type 2 is obtained from the sorted list. The search to locate where the new vertex should be inserted in the 2–3 tree has to be modified, since the *active* trapezoids have to be updated down to the y coordinate of the type 2 vertex to be inserted. If we update all the leaves of the tree, this will lead to a $\mathcal{O}(n \log r)$ algorithm. To obtain an $\mathcal{O}(r \log r)$ complexity, we resort to *lazy updating* of the tree. Before using a node of the 2–3 tree to decide where to branch, we update the leaves pointed at by these nodes (a maximum of two per node) down to the y coordinate of the type 2 vertex to be inserted. We then branch on the basis of the values given by the updated leaves. This will be correct, since the active *spans* never cross between type 2 vertices (a span is a sequence of trapezoids active between the type 2 or 3 vertex that created them and the type 2 or 3 vertex that terminates them). The number of leaves to be updated in this search is $\mathcal{O}(\log r)$. For each update, the cost could be up to $\mathcal{O}(n)$, since that number of trapezoids might have to be processed, but obviously the total number of updates is $\mathcal{O}(n)$. The number of leaves in the 2–3 tree at any given time is equal to the number of type 2 vertices already processed minus the number of type 3 vertices already processed, so it is $\mathcal{O}(r)$. Therefore, the cost of maintaining the 2–3 tree in this fashion is $\mathcal{O}(r \log r + n)$ and so is the cost of the whole of Algorithm 1. Since the triangulation step is $\mathcal{O}(n)$, the triangulation of simple polygons by this method is

$\mathcal{O}(r \log r + n)$. It should be noted then that it requires linear time for convex polygons, and for monotone polygons once they are rotated so that they are monotone along the y axis. This bound is not new, having been reached by Hertel and Mehlhorn [12]. The main differences here are that we use a trapezoid decomposition as an intermediary step and that the term r in our bound is the number of type 2 vertices, whereas in [12] it is the number of concave angles, which cannot be less.

4. Implementation

The preceding algorithms have been implemented in C under a UNIX operating system on a VAX 11/780. An interactive front end for the definition of the original polygons (and holes) and a back end for display to various devices (an Ikonas 3000-frame buffer and a Baush & Lomb DMP 40 pen plotter among others) have been added. Examples of the triangulations produced by the program are given in Figures 6 to 8. The code without the user interface and the output section adds up to about 2200 lines of code. The plot in Figure 9 shows the number of calls for key procedures as a function of the number of vertices in the polygon. It is clear that the rate of growth is as moderate as expected. The procedure **compare_keys** is perhaps the most relevant, since it gives the number of times a node in the 2-3 tree is examined. The procedures “angle,” “type,” and “which_side” are the only ones using a multiply operation. Table 1 gives the values plotted in Fig. 9.

Number of vertices	compare_					
	angle	keys	diagonal	horizontal	type	which_side
24	59	36	9	77	29	61
48	122	57	27	118	54	92
60	166	104	30	218	68	186
90	230	104	49	229	96	176
118	306	169	66	369	132	298

Table 1. Number of Calls per Function versus Polygon Size

5. Other Triangulations and Equivalent Problems

For the following problems, mainly triangulations of restricted classes of polygons, we give linear time transformations to the preceding problems and between each other.

5.1. Edge-Vertex Visibility

An edge of a polygon is **visible** from a vertex (and reciprocally) in a given direction iff there is a line segment parallel to that direction from the vertex to the edge entirely within the polygon. Solutions to the one edge-all-vertices visibility problem in all

directions or even to one vertex-all-edges visibility problem in all directions [5] have been found in $\mathcal{O}(n)$, but there is no solution yet better than $\mathcal{O}(n \log n)$ for the all edge-vertex visibility problem in one direction.

It is easily seen that the trapezoid list as described can be used to obtain the all edge-vertex visibility in $\Theta(n)$. The reverse is less obvious, since, to go from edge-vertex visibility to trapezoids, we need additional adjacency information. However, the following simple algorithm will provide it in $\mathcal{O}(n)$.

Algorithm 4a. Edge-Vertex Visibility to Trapezoid Decomposition

```

visibility_to_trapezoids(polygon)
if polygon not triangle or trapezoid do
begin
take first vertex of polygon with at least one visible edge;
mark vertex done if there is only one visible edge;
split polygon by horizontal line going from vertex to visible edge
into polygon1 and polygon2;
visibility_to_trapezoids(polygon1);
visibility_to_trapezoids(polygon2)
end

```

The actual intersections to split the polygons should not be computed, but stored as edge-vertex pairs, as before. We then can state:

Theorem 5.1. *The trapezoidization of a polygon can be obtained in $\Theta(n)$ time from its all edge-vertex visibility list, and reciprocally.*

The following algorithm will show that we can obtain edge-vertex visibility from a triangulation:

Algorithm 4b. Edge-Vertex Visibility from Triangulation

Input. A simple polygon with holes triangulated (in format described above). We assume each triangle is given in counterclockwise order.

Output. For each vertex a list of the edges visible from it in the x -axis direction.

Data Structure. *Edge_chain* – a sequence of complete or partial edges of the polygon which are x visible from an edge of the triangulation. The edge-chains are initialized to empty, are doubly linked, and are pointed at by both vertices of the edge.

```

begin
set all visible edge lists and edge chains to empty;
current = first vertex of polygon;
while polygon is not empty do
begin
if current is adjacent to only one triangle then
begin
delete triangle (in list for prev(current) and for next(current));

```

```

determine edge-chain visible from edge opposite to current;
this is either a merge of edge-chains from edges adjacent to current or
a split of one of the edge-chains
if opposite edge is a polygon edge then
begin
update edge-vertex visibility lists and delete portion of the
edge-chain matched against the polygon edge;
current = prev(current)
end
else current = next(current)
end
end

```

Analysis. Figure 10 illustrates the steps of this algorithm. By a theorem in [19], every polygon has an ear, so there is always a vertex that passes the test in the **while** loop. All such vertices in the original polygons are found in one pass over the list of vertices, and the ones created by the deletion process are either immediately before or immediately after the vertex deleted and, consequently, are found in constant time after the deletion. The edge-chains are kept in nonincreasing y order, so that the x projection inside the loop can be done in constant time. The only nonconstant time step is the update of vertex-edge visibility, which is done only once per visible edge-vertex pair, and their number is $\mathcal{O}(n)$. We then have the theorem:

Theorem 5.2. *The vertex-edge visibility of triangulated polygons (with holes) can be determined in $\Theta(n)$ time.*

5.2. Triangulation of a Star Polygon

The *kernel* of a polygon is the locus of points from which the whole polygon is visible. A *star polygon* is a simple polygon with a nonempty kernel. They have been extensively investigated, and efficient algorithms have been developed to decompose simple polygons into star polygons [2, 13]. The best upper bound known so far is $\mathcal{O}(n \log n)$ for a nonminimal decomposition. We now give our algorithm for triangulating a star polygon in $\Theta(n)$ time (see Fig. 11). The same upper bound, using a different algorithm, is achieved in [22].

Algorithm 5. Trapezoidization of a Star Polygon

Input. A star polygon as a list of vertices and a point of its kernel.

Output. A trapezoidized polygon in the format described above.

```

begin
for each vertex, determine if it is a peak -- that is to say a local maximum
    if above the kernel point and a local minimum if below;
for each peak trapezoidize up or down toward the kernel point
    until the kernel point is reached or a nonregular vertex is found;
trapezoidize the monotone polygon remaining by

```

```

merging the left and right list of vertices
end

```

Analysis. The first step of the algorithm takes $\mathcal{O}(n)$ time, and is similar to the classification of vertices described in Section 2.2. The second step, for each peak, is like the trapezoidization of a convex polygon, where it is enough to examine a left list and a right list to find the next vertex in y order. As soon as a nonregular vertex is found, the algorithm proceeds to the next peak. To prove that the first nonregular vertex found is on the left list or the right list, and not from another part of the polygon, assume (without loss of generality) that it happens when scanning from top to bottom. The type 2 vertex would then have a y coordinate greater than that of the kernel point. This means that there would be part of the polygon invisible from the kernel point on the other side of the type 2 vertex in contradiction with the definition of the kernel (see Figure 12).

After the pass over the peaks, there only remains a left and a right sequence of vertices, each monotone with respect to the y axis. These can be scanned as for a convex polygon. The running time of this algorithm then is linear as a function of the size of the polygon. The algorithm, as stated, needs a kernel point, but it is known that, given a simple polygon, its kernel can be determined in $\mathcal{O}(n)$ time [16]. This is true *a fortiori* for star polygons. We can now state:

Theorem 5.3. *A star polygon can be trapezoidized in $\Theta(n)$ time.*

Then from Theorems 3.2 and 5.3:

Theorem 5.4. *(also in [22]) A star polygon can be triangulated in $\Theta(n)$ time.*

The reverse transformation can be done trivially, since triangles are star polygons. Since any quadrilateral is also a star polygon, any two adjacent triangles can be merged, if it is necessary to reduce the number of star polygons in the decomposition. A further reduction can be effected by using a result from Chvatal [4] and exploited in [2], namely, that the vertices of a triangulated polygon can be three-colored, and the polygon decomposed into at most $\lfloor n/3 \rfloor$ star-shaped polygons by removing the internal vertices adjacent to all the vertices of a given color. The coloring can be done in $\mathcal{O}(n)$ time, in a straightforward way.

5.3. Triangulation of Monotone Polygons

A *monotone* polygon, following Garey et al.'s [11] definition, is such that it can be split into two lists of vertices in monotone order (without loss of generality, we assume it is with respect to the y axis, see Fig. 13). Thus, it can be trapezoidized like a convex polygon, in $\Theta(n)$ time, and by Theorem 3.2, triangulated with the same asymptotic complexity. Garey et al. showed that it can be triangulated directly in $\Theta(n)$ time. We can state:

Theorem 5.5. *[11] A monotone polygon can be triangulated in $\Theta(n)$ time.*

Since triangles are monotone polygons, the reverse, that we can decompose a triangulated polygon into monotone polygons in $\mathcal{O}(n)$ time, is trivially true.

5.4. Triangulation of Spiral Polygons

A *spiral* polygon is a polygon having at most one sequence of adjacent reflex vertices. Feng and Pavlidis [6] give an algorithm to decompose a polygon into a (not necessarily optimal) set of spiral polygons.

A linear-time triangulation of a spiral polygon is easily obtained by noting that (unless it is a triangle) it has an ear at each end of the reflex chain, and that removing these ears leaves a spiral polygon with two fewer vertices (see Fig. 14). This then gives the theorem:

Theorem 5.6. *A spiral polygon can be triangulated in $\Theta(n)$ time.*

The reverse information, from triangles to spiral polygons is immediate, since triangles themselves are spiral polygons, and any two adjacent triangles form a spiral polygon.

6. The Triangulation-Linear Class

We have given here new linear-time algorithms for the following transformations:

- trapezoid decomposition to triangulation,
- star polygon decomposition to triangulation.

Other linear transformations, shown in Fig. 15, are more straightforward or already well known. All the algorithms given take $\mathcal{O}(n)$ space. All these problems, therefore, form an equivalence class, which can be called the *triangulation-linear* class, such that the members of the class are linearly related to each other. Consequently, if any algorithm leads to a solution to one of them in $F(n) \geq \mathcal{O}(n)$, then it leads to a solution to all of them in $F(n)$, and if any lower bound $G(n) \geq \Omega(n)$ is proved for one of them, it applies to all of them. The next obvious question is whether the lower bound for any of them is greater than $\Omega(n)$. Unfortunately, we cannot answer this question for now, but the existence and extent of the class should make the answer easier to find. All these problems share an upper bound of $\mathcal{O}(n)$, and it also should be noted that for the restricted polygons used (convex, star, monotone—if the axis is given, spiral, trapezoid) their characteristic property can be checked in $\mathcal{O}(n)$ time. The monotonicity of a polygon can also be tested in $\mathcal{O}(n)$ even if the axis is not given [20].

The next question is to find other members of the class. They are problems that seem to need as much information as a triangulation, but cannot be used to obtain a triangulation in linear time, unless it is its upper bound. As mentioned above, every polygon has at least an ear (three consecutive vertices that form a triangle in its triangulation). In fact, except for a triangle, each polygon has at least two

nonoverlapping ears. It is obvious that an $\mathcal{O}(n)$ scan of the triangulation can produce the list of ears of the polygon. The converse, namely, that the list of nonoverlapping ears of the polygon can be used to produce a triangulation in $\mathcal{O}(n)$ time, is true only if triangulation is $\Omega(n)$. If it is true, we can triangulate a polygon by adding new vertices in the middle of each edge at an ε distance from the edges, and give these vertices for the set of nonoverlapping ears. If this could be done in $\mathcal{O}(n)$ time, and if from the ears we could obtain the triangulation in $\mathcal{O}(n)$ time, the whole triangulation process would be $\mathcal{O}(n)$.

7. Conclusion and Further Problems

The two main results in this paper are an algorithm and its implementation to triangulate simple polygons with holes in $\mathcal{O}(n \log n)$ with simple and safe basic operations, and the establishment of a class of problems linearly related to the triangulation of a polygon. Most of the problems in this class are decomposed into a restricted class of polygons (convex, star, spiral, monotone, trapezoid). The decompositions obtained are generally not unique and not optimal.

There are a number of open problems related to these decomposition problems. The optimal (in number of elements) decomposition of polygons with holes is known to be NP-complete or NP-hard in most cases (trapezoids [1]; convex polygons [17]; monotone, star, spiral polygons [13]). A more tractable question is to determine the complexity of the optimal decomposition into these various polygons given the triangulation, if no holes are allowed.

Another research direction is to investigate the characteristic properties of the triangulations obtained by these various routes. This would help decide the suitability of different algorithms of similar complexity for a particular application.

Acknowledgment

The authors would like to thank Avi Naiman for implementing the 2–3 tree manipulation functions and some of the user-interface functions, and for discussing many related issues.

References

- [1] Asano, T., and Asano, T. Minimum partition of polygonal regions into trapezoids. In *Proceedings of the 24th Annual Symposium on the Foundations of Computer Science* (Tucson, AZ, Nov 7–9), IEEE, Los Angeles, 1983, pp. 233–241.
- [2] Avis, D., and Toussaint, G. T. An efficient algorithm for decomposing a polygon into star-shaped polygons. *Pattern Recogn.* 13, 6 (1981), 395–398.

- [3] Chazelle, B., and Dobkin, D. Decomposing a polygon into its convex parts. In *Proceedings of the 11th Symposium on Theory of Computing* (Atlanta, GA, April 30–May 2), ACM, New York, 1979, pp. 38–48.
- [4] Chvatal, V. A combinatorial theorem in plane geometry. *J. Comb. Theory Ser. B* 18 (1975), 39–41.
- [5] El Gindy, H., and Avis, D. A linear algorithm for computing the visibility polygon from a point. *J. of Algo.* 2, 4 (June 1981), 186–197.
- [6] Feng, H., and Pavlidis, T. Decomposition of polygons into simpler components: Feature generation for syntactic pattern recognition. *IEEE Trans. Comput.* C-24 (June 1975), 636–650.
- [7] Fiume, E., Fournier, A., and Rudolph, L. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultra-computer. *ACM Comput. Graph.* 17, 3 (July 1983), 141–150.
- [8] Foley, J., and van Dam, A. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Reading, Mass., 1982.
- [9] Fuchs, H., Poulton, Paeth, A., and Bell, A. Developing PIXEL-PLANES, a smart memory-based raster graphics system. In *1982 Conference on Advanced Research in VLSI*, MIT Press, Boston, Mass., 1982, pp. 137–146.
- [10] Fussell, D., and Rathi, B. D. A VLSI-oriented architecture for real-time raster display of shaded polygons. In *Proceedings of Graphics Interface '82*, National Research Council of Canada (Toronto, Ontario, May 17–21), 1982, pp. 373–380.
- [11] Garey, M. R., Johnson, D. S., Preparata, F. P., and Tarjan, R. E. Triangulating a simple polygon. *Inf. Proc. Lett.* 7, 4 (June 1978), 175–179.
- [12] Hertel, S., and Mehlhorn, K. Fast triangulation of simple polygons. In *Proceedings of the 1983 International Conference on the Foundations of Computer Science* (Tucson, AZ, Nov. 7–9), IEEE, Los Angeles, 1983, pp. 207–218.
- [13] Keil, J. M. Decomposing polygons into simpler components. Tech. Rep. 163/83. Dept. of Computer Science, Univ. of Toronto, 1983.
- [14] Kirkpatrick, D. G. Optimal search in planar subdivisions. *SIAM J. Comput.* 12, 1 (Feb. 1983), 28–35.
- [15] Lee, D. T. Shading of regions on vector display devices. *ACM Comput. Graph.* 15, 3 (July 1981), 34–44.
- [16] Lee, D. T., and Preparata, F. P. An optimal algorithm for finding the kernel of a polygon. *J. ACM* 26, 3 (July 1979), 415–421.

- [17] Lingas, A. The power of non-rectilinear holes. In *The 9th International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science 140. Springer-Verlag, New York, 369–383.
- [18] Lloyd, E. L. On triangulations of a set of points in the plane. In *Proceedings of the 18th Annual Symposium on the Foundation of Computer Science* (Providence, RI, Oct. 31–Nov 2), IEEE, Los Angeles, 1977, pp. 228–240.
- [19] Meisters, G. H. Polygons have ears. *Amer. Math. Monthly* 82 (1975), 648–651.
- [20] Preparata, F. P., and Supowit, S. Testing a simple polygon for monotonicity. *Inf. Proc. Lett.* 12, 4 (Aug. 1981), 161–164.
- [21] Schachter, B. Decomposition of polygons into convex sets. *IEEE Trans. Comput.* C-27, 11 (Nov. 1978), 1078–1082.
- [22] Schoone, A. A., and van Leeuwen, J. Triangulating a star-shaped polygon. Tech. Rep. RUV-CS-80-3, Univ. of Utrecht, Holland, 1980.
- [23] Shamos, M. I. Geometric complexity. In *Proceedings of the 7th Annual Symposium on Theory of Computing* (Albuquerque, N.M., May 5–7), ACM, New York, 1975, pp. 224–233.
- [24] Watkins, G. S. A real-time visible surface algorithm. Tech. Rep. UTEC-CSc-70-101, Computer Science Dept., Univ. of Utah, 1970, NTIS AD-762 004.

Received November 1983; revised October 1984; accepted October 1984.