EUROGRAPHICS '94 / M. Dæhlen and L. Kjelldahl (Guest Editors), Blackwell Publishers

Volume 13, (1994), number 3

Triangulating multiply-connected polygons: A simple, yet efficient algorithm.

Remi P. Ronfard *Jarek R. Rossignac*
IBM T.J.Watson Research Center
P.O.Box 704, Yorktown Heights, NY 10598

Abstract

We present a new, simple, yet efficient algorithm for triangulating multiply-connected polygons. The algorithm requires sorting only local concave minima (sags). The order in which triangles are created mimics a flooding process of the interior of the polygon. At each stage, the algorithm analyses the positions and neighborhoods of two vertices only, and possibly checks for active sags, so as to determine which of five possible actions to take. Actions are based on a local decomposition of the polygon into monotonic regions, or *gorges* (raise the water level in the current gorge, spill into an adjacent gorge, jump to the other bank of a filled gorge, divide a gorge into two, and fill a gorge to its top). The implementation is extremely simple and numerically robust for a large class of polygons. It has been tested on millions of cases as a preprocessing step of a walkthrough and inspection program for complex mechanical and architectural scenes. Extensive experimental results indicate that the observed complexity in terms of the number of vertices, remains under $O(N^{\frac{3}{2}})$ in all cases.

1 Introduction

Triangulation is a fundamental operation in computational geometry and has been studied extensively in an attempt to reduce its algorithmic complexity, i.e. the running time of triangulation as a function of the number of vertices N in the polygon [7] [5] [6] [1]. Triangulation is particularly important in geometric modeling and in graphics. In these applications, the performance of triangulation algorithms cannot be evaluated solely in terms of their algorithmic *worst-case* complexity. More important is the average *expected* behaviour for polygons with a *reasonable* number of edges. Several efficient triangulation algorithms have been proposed for polygons that are *simply-connected* (without holes) [3]. Unfortunately, many computer-aided design systems produce multiply-connected faces that need to be triangulated for efficient rendering and for other downstream applications.

We present in this paper an efficient algorithm, called *flooding*, which works for both simply and multiply connected polygons. The remainder of this section defines the domain, i.e., the class of polygons properly triangulated by the flooding algorithm and their representation accepted as input format by the algorithm. Section 2 positions the proposed solution in the context of previous publications. Section 3 introduces a suitable vocabulary for presenting the

algorithm, in terms of the metaphor of *flooding* a polygon with water. Section 4 describes the algorithm (at an intuitive level) in terms of a finite state machine with only 6 possible states and five transition operations. Section 5 provides implementation details. Section 6 discusses the correctness of the algorithm, and addresses the handling of degenerate (invalid) input models or of topological errors resulting from numerical inaccuracies. Finally, the algorithmic complexity is analyzed in Section 7.

We define a polygon to be a bounded, connected subset of the plane, such that it is equal to the interior of its closure and that it has a finite piecewise linear (nowhere dense) boundary. The boundary may be decomposed into a finite set of *vertices* and *edges* as follows. Vertices are the points with non-linear neighborhoods. Edges are the maximally connected components of the boundary minus its vertices.

The boundary of a polygon may be partitionned into maximally connected components called loops. There exists a unique circular order of all the edges within a loop. The order is obvious at manifold vertices. The edge orientation defines the start-vertices and end-vertices. The *next* edge is the one starting at the end-vertex of the current edge. At non-manifold vertices there are two or more such edges. We chose as *next* the first one encountered while rotating in a clockwise manner around the end-vertex. Given the above convention, a polygon has a unique representation (as a set of loops, each loop being a circular ordering of vertex references) which is always valid. In this paper, we address the issue of triangulating a polygon represented using our conventions. The triangulation produces a list of triangles T and internal edges I such that:

a) Internal edges of I are relatively open line segments contained in the polygon and connect vertices of the polygon (triangulation does not add artificial vertices),

b) Internal edges and triangles partition the generalized polygon (they are pairwise disjoint and their union is the entire polygon).

2 Previous work

Triangulation may be performed by first decomposing the polygon into simpler parts and then by triangulating each part with an efficient, special-purpose method. As an example, polygon decomposition into convex parts [9] yields an efficient method of triangulation, because *any* vertex in a convex polygon can be joined by internal lines to *all* other non-adjacent vertices to produce a correct triangulation. Another interesting approach consists in adding horizontal internal edges, so as to decompose the polygon into a union of trapezoids. Since trapezoids are trivially triangulated, this again provides an efficient approach [2] [4], which has been recently improved to a linear complexity algorithm [1]. A more general (monotonic) decomposition method has been described by Lee and Preparata [3], as a special case of what they call *regularization* of a planar graph. The regularization algorithm finds a set of monotonic regions partitioning both the interior and the exterior of any simple polygon in $N \log N$ time and N space. An efficient triangulation of monotonic polygons has been described by Garey and his colleagues in 1978 [5] and is also discussed in [4].

The new flooding algorithm applies to all types of polygons, whether simply or multiply connected. Flooding generalizes the monotonic decomposition of [3] and the monotonic tri-

angulation of [5], but does not require building their complex auxiliary data-structures. It is similar in spirit to the sweep-line algorithm of [6], but sweeps only one gorge at a time.

3 The flood metaphor.

A coordinate system (a direction, the *vertical*, and an origin) is chosen and used to define the *height* for vertices. A polygon is *monotonic* if its boundary has exactly one maximum and one minimum. A polygon may be decomposed into monotonic regions: its *gorges*. (Such a decomposition is in general not unique.) Since a gorge is monotonic, its boundary can be split into two parts, its *left and right banks*.

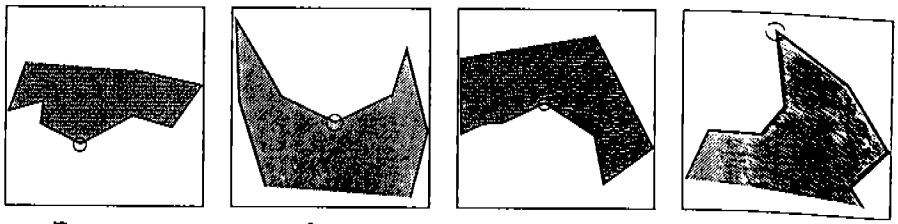


Figure 1: Non-monotonic vertex types.

As illustrated in figure 1, there are four types of special vertices: a convex local minimum is called a *bottom*, a concave local minimum is called a *sag*, a concave local maximum is called a *top*, and a convex local maximum is called a *ceiling*. Note that no distinction needs to be given to non-manifold vertices. A *wedge* is a chain of adjacent triangles sharing a common vertex, called the apex of the wedge.

The internal data-structure used by the flooding algorithm is an array of vertices and a list of loops, each stored as a doubly-linked list of vertex references. As the algorithm progresses, loops may be split or merged, but there is one *active loop* at any moment.

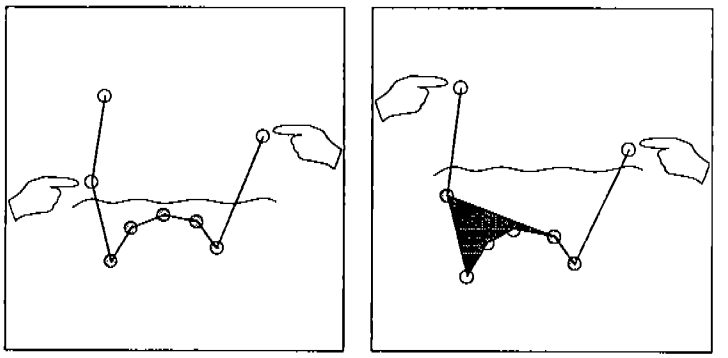
The *active chain* is a connected subset of the active loop. Its starting and ending vertices are called respectively *Left* and *Right*. The vertex preceding Left is called NextLeft. The vertex following Right is called NextRight. (NextLeft and NextRight are indicated by fingers of left and right hands in the following figures.) The active chain always forms a concave line and is bounded by a bottom vertex of the active loop. The Right and Left vertices are on the opposite sides of a gorge. Initially, the active chain contains only one vertex, the lowest bottom in the external loop of the polygon, which is the Left *and* the Right vertex of the active chain.

In the simplest situation, the flooding algorithm fills that gorge by removing bottom triangles and by extending the active chain to its neighbor edges. Basically, the height of NextLeft and NextRight are compared. Suppose, without loss of generality that NextLeft is lower. The vertices of the active chain that are visible from NextLeft (and connected to NextLeft, by either external or internal edges) define a wedge of triangles to be removed from the polygon. The active loop is adjusted and NextLeft becomes Left. This operation

is called a *raise*, and can be iterated in a monotonic polygon, which it will triangulate in $O(N)$ time [3] [5] [6] [7]. The main contribution of our method consists in the four other operations (bridge,fill,spill and jump), which together with the *raise*, will triangulate an arbitrary triangle, in a similar fashion.

4 High-level description of the algorithm.

In the *pre-processing* stage, we extract all sags and sort them in height. (Equal height conflicts are resolved using lexicographics order of both the X and Y coordinates of the vertices.) The sorted list of sags reduces the cost of searching for the lowest sags inside the current gorge.



Before a *raise* After a *raise*

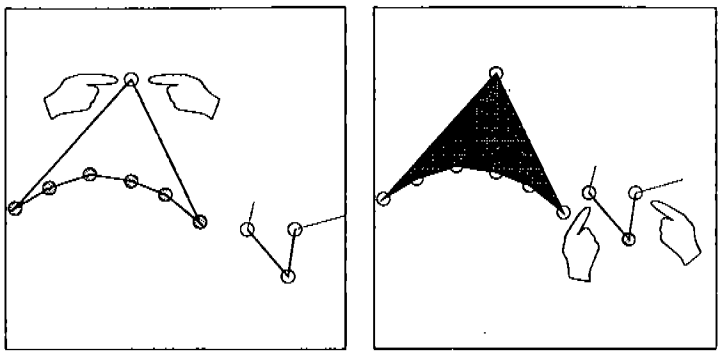
Figure 2: Raise.

Each sag will eventually be connected to a top or a regular vertex by an internal edge. Such a *bridge* operation may divide the active loop or connect it to another loop creating two gorges. The algorithm will pursue (and fill) one of these gorges. The bottom of the other gorge will be put on a *to-do* list of pending bottoms. When the top of a gorge is reached (the gorge is full), the flooding algorithm checks the list for bottoms of unprocessed gorges. (Note that a pending bottom added to the *to-do* list during a *bridge* operation can also be processed as a result of spilling around a hole, in which case it will *not* require further actions.)

Each vertex has pointers to its immediate right and left neighbours in a loop, plus an additional *jump* pointer, initialized to null. The purpose of *jump* is to point directly to the other bank of a previously filled gorge, that was momentarily put on hold to pursue a spill.

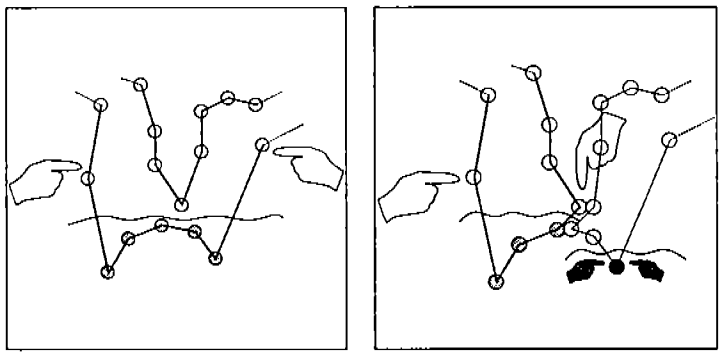
Flooding may be viewed as a *finite-state automaton* with only the following five transitions (i.e., actions):

- **Raise.** Assume (without loss of generality) that NextLeft is below NextRight. If NextLeft is above Left (the edge goes up and Left is not a top), we search the sorted list of sags for the lowest sag above Left and Right and below NextLeft in that gorge. If no such sag exists, one can triangulate a portion of the polygon by constructing interior edges between NextLeft and the vertices of the active chain visible from NextLeft (cf. fig. 2).



Before a *fill* After a *fill*

Figure 3: Fill.



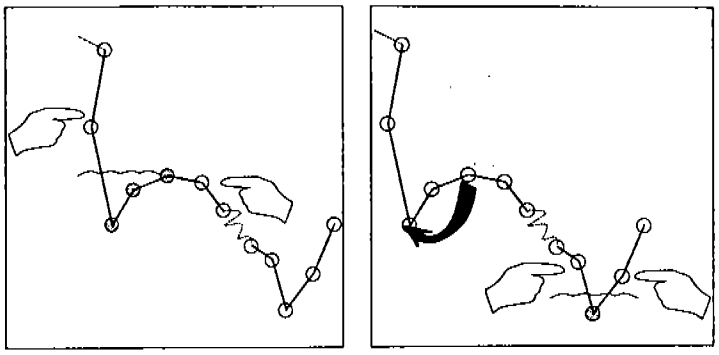
Before a *bridge* After a *bridge*

Figure 4: Bridge.

- **Fill.** If NextLeft and NextRight are the same, internal edges can be created from this vertex to all non-adjacent vertices in the loop triangulating the remainder of the polygon (figure 3). After a *fill*, the Left and Right pointers are reset to the next non-processed pending bottom.

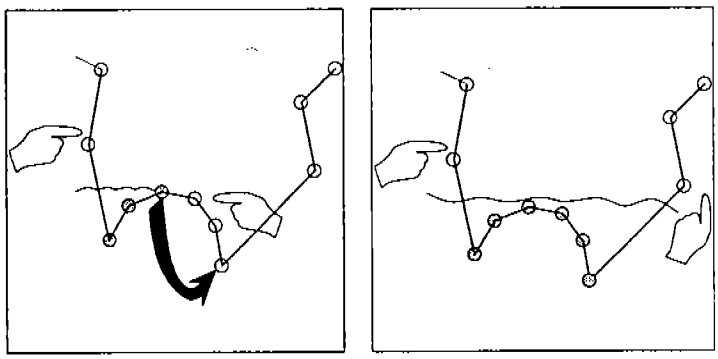
- **Bridge.** If a sag is found during the above check, we create a *bridge* of two internal edges from the sag to the highest vertex in the active chain. After a Bridge, the chain is either split into two subchains, one of which remains active, while the other one remains attached to a new *pending bottom* (cf fig. 4).

- **Spill.** If Left (respectively Right) is a top, and has a null *jump* pointer), we set its jump pointer to Right (respectively Left) on the opposite bank of the gorge for when we are done filling the adjacent gorge, and then follow the chain towards the left (respectively right), searching for the next bottom *not matched* by a sag encountered along the search path. This new bottom becomes the start of a new gorge-flooding process (it becomes



Before a *spill* After a *spill*

Figure 5: Spill.



Before a *jump* After a *jump*

Figure 6: Jump.

the Left and the Righ of a new active chain, see figure 5). The requirement for skipping as many bottoms as sags encountered during the search guarantees that we will not attempt to back-track a spill, which would create an infinite loop.

- **Jump.** If the Left (respectively Right) pointer is a top, and its *jump* pointer is not null, we jump to the other bank (i.e., reset it to the value in the jump pointer), as seen on figure 6.

The entire flooding process is illustrated in figures 7 and 8, which show one example of triangulation, at several important intermediate steps of the algorithm (figure 7), with all triangles numbered according to the order in which they have been flooded (in figure 8).

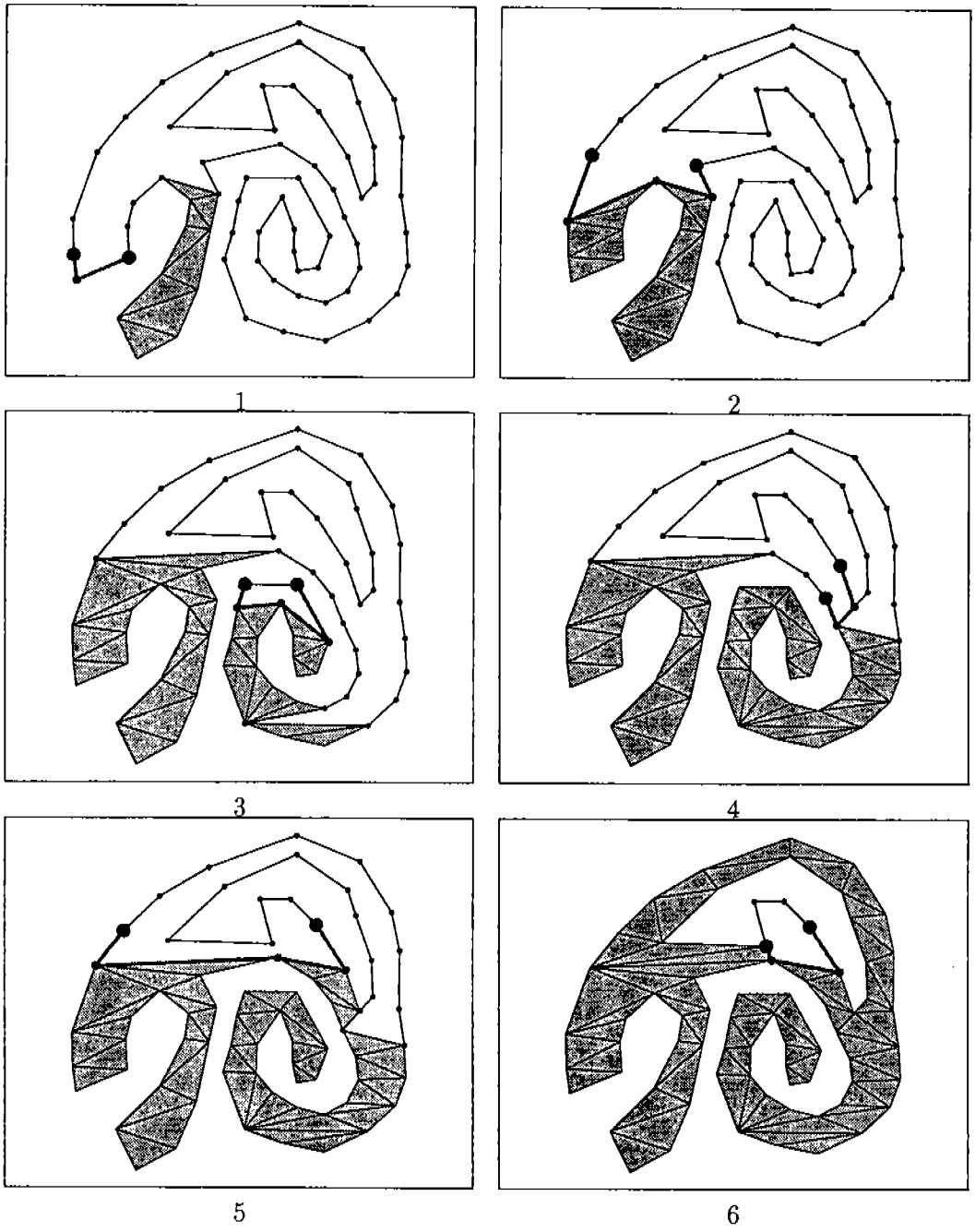


Figure 7: Intermediate steps. Each figure shows the result of a series of actions, followed by numbers indicating which triangles have been created and removed: (1) Raise 0-7, Spill; (2) Raise 8-12, Jump; (3) Raise 13-18, Bridge, Raise 19-26, Spill, Raise 27-30, Jump. (4) Raise 31-33, Fill. Raise 34-41, Bridge; (5) Raise 42-45, Jump; (6) Bridge, Raise 46-52, Spill, Raise 53-62, Jump, Raise 63-64, Fill.

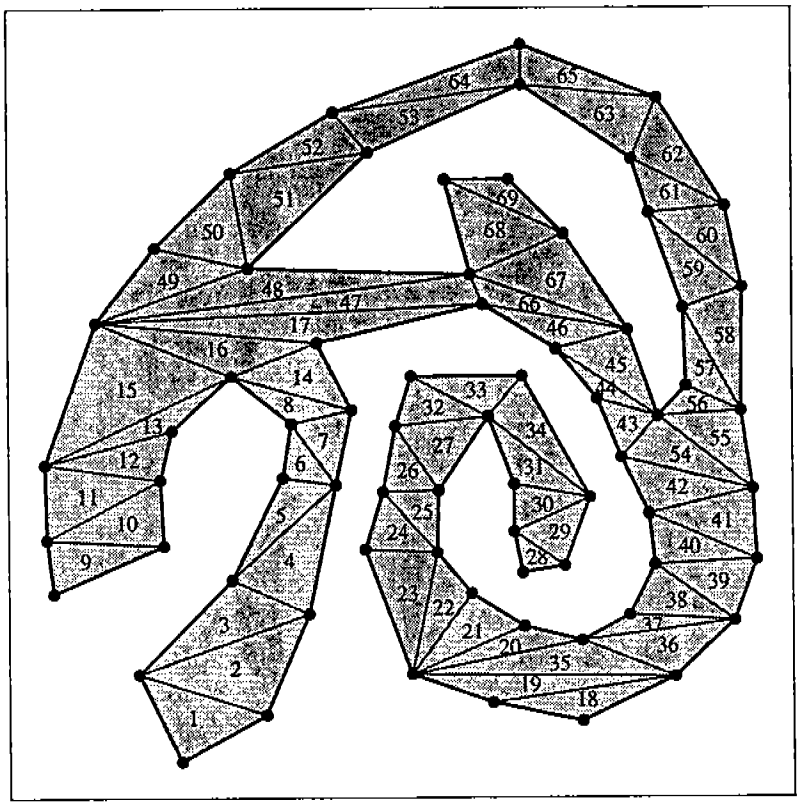


Figure 8: Flooding triangulation example. The numbers indicate the order in which triangles have been removed.

5 Implementation details

The control mechanism for our finite-state implementation of the method is based on the decision tree summarized in figure 9. In order to control numerical stability and handle degenerate cases, the number of geometric routines has been kept at a minimum. Our implementation consists of just three geometric predicates, called *convex*, *below* and *inside*.

Convex. A predicate $convex(p_1, p_2, p_3)$ is used to decide whether vertex p_2 between p_1 and p_3 is convex. Given the orientation of p_1 , p_2 and p_3 in their loop, this solves the question whether triangle (p_1, p_2, p_3) lies inside or outside the polygon.

Below. A predicate $below(p_1, p_2)$ is used to compare the heights of a pair of vertices, with the convention that vertices with equal height use the other coordinate for comparison (this defines a lexicographic ordering of vertices, based on both their X and Y coordinates in the plane; we can still represent this ordering as a *height* if we give an imaginary tilt on all horizontal lines).

Inside. A predicate $inside(S, A, B)$ is needed to determine whether a given vertex S (a sag) lies inside the quadrangle formed by the Left and Right pointers and their fingers NextLeft and NextRight (fig. 4). This is obviously a sufficient condition for a sag S to be inside the active gorge. For sags whose heights have been bracketed between the current water level L_1 and the next water level L_2, this will be a necessary condition also.

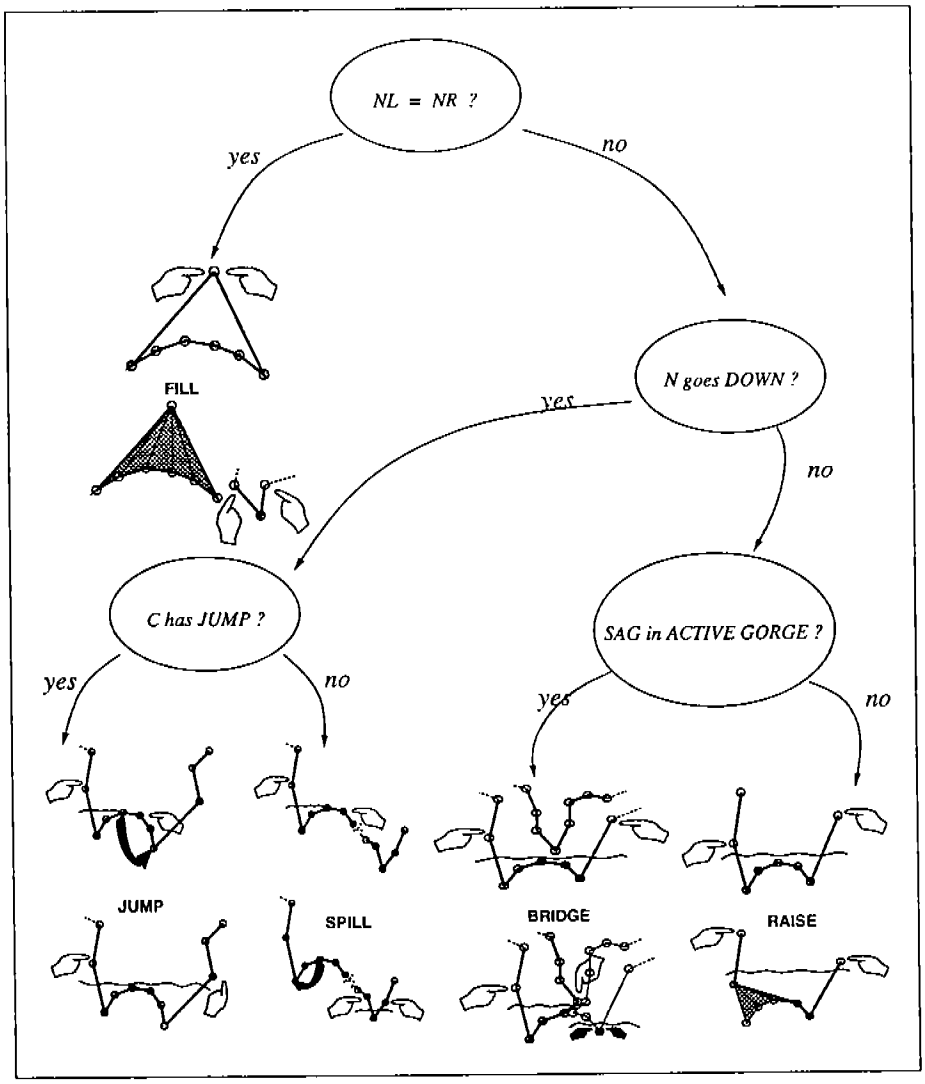


Figure 9: Decision tree for flooding actions. Next is the lowest of NextLeft (NL) and NextRight (NR); C is the corresponding left or right end of chain (Left or Right).

6 Correctness of the algorithm.

Although a formal proof of our algorithm is difficult, we can nevertheless show that if the polygon is valid, then (a) the algorithm produces a valid set of triangles, and (b) the resulting triangulation covers the input polygon completely. In order to prove (a), we first remark that all vertices between the Left and Right pointers (if any) are concave. Only the Left and Right ends of the active chain can ever be convex, which is why we claim that our algorithm generalizes Garey et al. [5]. As a result, the local configuration in the active chain is always similar to a monotonic polygon, which we can triangulate correctly as long as no other edges are contained in it. Because we detect sags, this can never happen, and all triangles are therefore *valid*. Secondly, as we triangulate a polygon, we update the boundary of the non-triangulated region, by *chopping off* triangles as we progress. We now must explain how this process eventually leads to a set of *empty* regions, and thus prove (b). Intuitively, the reason is that there is only one way of closing the active chain, which is to go through *all* vertices in the initial bottom's loop. It is then easy to prove that when this happens, the remaining region enclosed will be completely triangulated by the final *fill*. If the initial bottom's loop is monotonic, i.e. without tops nor enclosed sags, this intuition is easily proved, as in [5]. If the loop encloses sags, but has no tops, it will be divided into several components, each of them monotonic, and therefore each of them completely triangulated. The more difficult case comes with *tops* and spilling. The active chain is temporarily disrupted after a *spill*, but this is necessarily followed by a corresponding *jump* from the opposite direction, because each top has exactly two different banks. Although this is quite difficult to prove formally, it intuitively shows that the disrupted chain is always restored, and therefore no vertices are left behind (see [8] for more detail). As a consequence, flooding will produce a valid triangulation when loops are valid. When this is not the case, our finite-state machine implementation provides at least to direct ways to track down errors and degeneracies: forbidden transitions and forbidden states (see fig. 10).

	monotonic	ceiling	sag	top	bottom
monotonic	R,B,F	X	X	X	R,B,F
ceiling	X	X	X	X	X
sag	X	X	X	X	X
top	S,J	X	X	X	S,J
bottom	R,B,F	X	X	S,J	R,B,F

Figure 10: Vertex combinations and states. States are raise (R), fill (F), bridge (B), spill (S), jump (J) and forbidden (X).

Because of its finite state implementation, our algorithm either ends successfully or an error is detected (forbidden state or transition). As an example, consider invalid polygons represented in fig. 11. In 11-A, the Right and Left pointers are identical, and classified as a *sag* because the polygon intersects itself. In 11-B, the Left pointer is classified as a *ceiling*, not as a top, so that spilling should not occur, and an error is detected for the same reason.

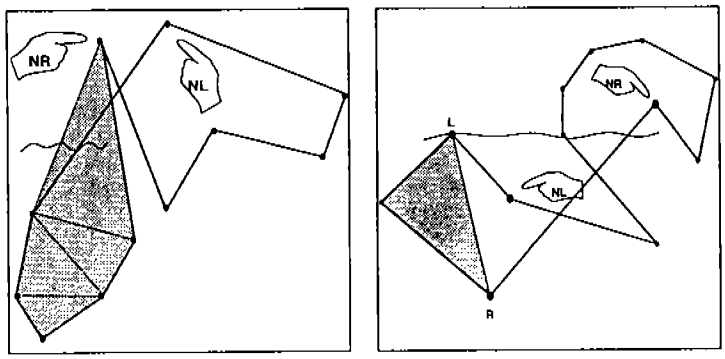


Figure 11: Invalid polygons and error detection. A,B: Invalid polygons.

In practice, we triangulate both the polygon and its complement, for verification purposes. We leave as an open conjecture that this detects all invalid or degenerate polygons.

7 Complexity analysis

Let us first describe worst-case costs for all relevant steps of the algorithm. Identifying bottoms and sags requires two *below* and one *convex* predicates to be evaluated. This is therefore linear in the total number N of vertices. Sorting sags can be done in $S\log(S)$ time, where S is the number of sags. Finding the next sag is linear in the number of sags, because sags are sorted. For each gorge, the worst case is when all sags between the current and next water levels have to be tested, and they are all outside. The *inside* test is then performed a maximum number of SG times in all, where G is the number of gorges. As a conclusion, the asymptotic behaviour of the flooding algorithm can be predicted to be $O(N) + O(SG)$, therefore $O(N^2)$ in the worst case.

In practice, we have observed almost-linear behaviour in many different cases, and an actual worst-case complexity of $O(N^{\frac{3}{2}})$. We base this claim on a series of experiments, described in [8], in which we fitted a log-linear model to the running times T vs. number of vertices N, in a variety of practical cases, i.e. $T = k\,N^A$ where A is the exponent. Our findings are that the exponent varies between 0.9 and 1.4 in all cases (see [8]) . It should also be noted that the connectivity type (number of holes) does not in itself add to the complexity, because holes are taken into account naturally (as sags) in our framework.

8 Conclusion

We have presented an algorithm for triangulation of a general class of multiply connected polygons, and described a compact implementation of it as a finite state machine. The implementation is remarkably robust because numerical errors generate *impossible* transitions, which can be checked and reported at no cost. The asymptotic behaviour of our approach

depends on the relative number of monotonic and *special* vertices (sags, tops, ceilings and bottoms). If the proportion of special vertices decreases with the total size of the polygon, as is usually the case in practical situations (smooth, faceted surfaces, manufactured objects), the method becomes linear in the total number of vertices. On average, it has been found to be $O(N^{\frac{3}{2}})$, although its theoretical worst-case behaviour can be predicted to be $O(N^2)$. One limitation in our approach is that self-intersecting polygons can only be reported as *invalid input*, but not triangulated. On the other hand, within the class of non-intersecting polygons, we are able to deal efficiently with an arbitrary number of holes, in a very natural way.

9 Acknowledgements

We would like to thank V. Srinivasan and V.T. Rajan for useful comments on this work.

References

[1] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Computational Geometry*, 6:485–524, 1991.

[2] B. Chazelle and J. Incerpi. Triangulation and shape complexity. *ACM trans. on Graphics*, 3:135–152, 1984.

[3] Lee D.T. and Preparata F.P. Location of a point in a planar subdivision and its applications. *SIAM J. on Computers*, 6:594–606, 1977.

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

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

[6] S. Hertel and K. Melhorn. Fast triangulation of simple polygons. In *Conference on Foundations of Computer Science Theory*, pages 207–218, New-York, 1983.

[7] J. O'Rourke. *Art gallery theorems and algorithms*. Oxford University Press, New York, 1987.

[8] R. Ronfard and J. Rossignac. Triangulating multiply-connected polygons: A simple, yet efficient algorithm. Technical report, IBM Research, Yorktown Heights, 1994.

[9] S.B. Tor and A.E. Middleditch. Convex decomposition of simple polygons. *ACM trans. on Graphics*, 3(4):244–265, 1984.