An Implementation of a Near-Linear Polygon Triangulation Algorithm for General Polygons

Abstract

In 1991 Seidel found a practical algorithm for triangulating simple polygons with an expected running time of $O(n \log^* n)$. This paper describes an implementation of his routine, and shows how the theoretical time bound compares to the experimental. Several generalizations and optimizations of his routine are discussed, and the final result is an algorithm that can triangulate any set of overlapping and self-intersecting polygons and lines in the plane with near-linear expected running time. The implementation is completed with a set of functions that will graphically display any step of the algorithm.

1 Introduction

The problem of triangulating a simple polygon is an old and important computational problem. It can be simply stated as finding all the diagonals from one vertex to another in a given polygon that will partition the polygon into a set of triangles. A *simple polygon* is a polygon that has no self-intersections, n edges and n vertices. Polygon triangulation is used in many areas, and the search for optimal algorithms to compute it has a long history.

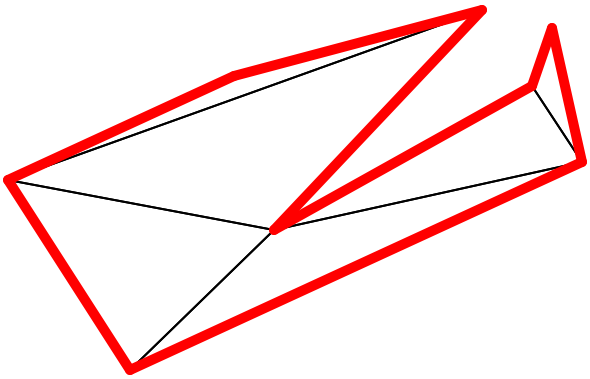


Figure 1: A triangulated polygon. The red edges are edges of the polygon, the black lines are the diagonals introduced.

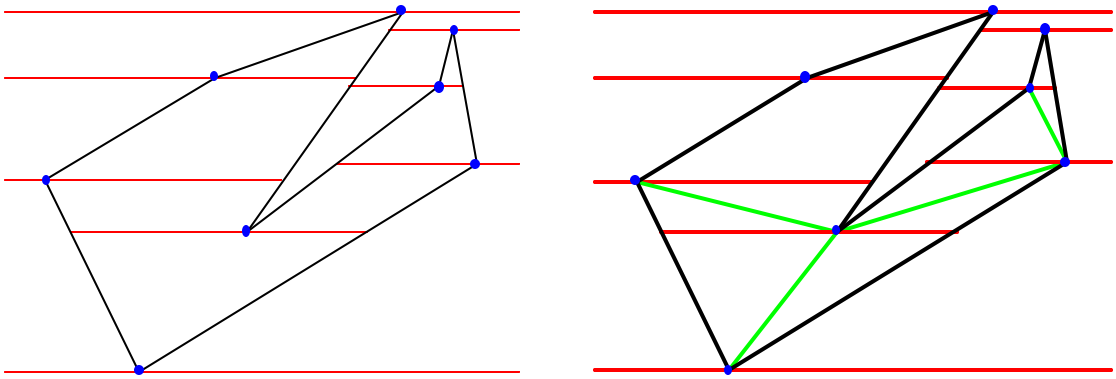
Polygon triangulation is an essential problem in computational geometry. Working with a set of triangles is in most cases faster than working with the entire polygon, as the answers to queries often can be found by searching only locally. Many algorithms assume that the polygon is already triangulated, and several would not be possible without the triangulation. Given a triangulation, other partitions such as convex or star polygons can easily be found in linear time. Some algorithms that rely entirely upon partitioned polygons are character recognition [O'Rourke], shading [Fournier & Montuno] and shortest path [Guibas et al.].

Given its importance, a lot of effort has been put into finding a fast polygon triangulation routine. The naive recursive triangulation of a polygon runs in time $O(n^3)$ by cutting ears from the polygon. $O(n^2)$ algorithms have been known since at least 1911 (Lennes; see [O'Rourke]). But it wasn't until 1978, when Garey et al found an $O(n \log n)$ algorithm, that real work started in this field. A variety of new algorithms were found, pushing the limit further and further down. In 1984 it was shown that given a

trapezoidation of a polygon, a triangulation could be obtained in linear time (Fournier & Montuno, Chazelle & Incerpi), and further work focused on finding an efficient way of computing such a trapezoidation. However, the outstanding open problem in computational geometry, that of the lower time bound of triangulation, was not solved until Chazelle found a purely linear algorithm in 1990. This algorithm is very complex, and has to date not been implemented. In 1991, Seidel found a practical randomized $O(n \log^* n)$[1] algorithm, which is the one I have chosen to implement. It should also be mentioned that in 2000 Amato et al. found a simpler linear algorithm with far less overhead than Chazelle's, by using randomization as in Seidel's algorithm. This would still be a formidable job to implement with all its details, and the search for a practical and fast linear triangulation method is still not over. For further information on the development of triangulation, see for instance [O'Rourke].

2 Trapezoidation

A *trapezoidation* is formed by extending a horizontal[2] ray in each direction from every vertex, and stopping the ray as soon as it hits another edge. This will divide the plane into regions by edges and rays, and each such region is a *trapezoid*[3]. A trapezoid has a horizontal upper and lower boundary, and an edge or a piece of an edge as a boundary on either side. Either the upper or lower boundary may have zero length, in which case the trapezoid will be a triangle. See figure 1 (a). The set of trapezoids obtained in this manner is called a trapezoidation, and is unique for any given polygon.



(a) (b)

Figure 2: A trapezoidation of a polygon, (a) without and (b) with trapezoid diagonals

Given the trapezoidation of a polygon, a triangulation can easily be computed in linear time. Let us consider only those trapezoids that are inside the polygon, as these are the ones we will want to triangulate. First of all, note that each trapezoid will be empty, i.e. there will be no vertices or edges inside it. Further note that the upper and lower

[1] Log* will be properly defined in section 6. For now, it suffices to note that for large n, log* n << log n.
[2] The choice of the direction of the rays is arbitrary, as long as all are parallel.
[3] Technically, regions bounded by $\pm\infty$ are not trapezoids, but will be considered to be in this paper.

boundary of each trapezoid are created by the extension of rays from vertices, such that each trapezoid has one bounding upper and one bounding lower vertex. Lower is here defined lexicographically, i.e. of two vertices with the same y-coordinate, the rightmost one will be considered lower. This is assumed for all vertices throughout this paper, and assures us that no edge is being treated as if it was horizontal, and that no trapezoid will have more than one upper and one lower bounding vertex. In some cases the two bounding vertices will be on the same side of the trapezoid, and in other they may be on opposite sides, or in the middle. Whenever the two vertices are not along the same edge, one can draw a diagonal between them without intersecting any edges since each trapezoid is empty. The addition of these diagonals is illustrated in figure 1 (b). The resulting polygons bounded by edges and diagonals are monotone mountains, see [O'Rourke] for a proof of this.

Monotone mountains have one edge called the base, which extends from the uppermost point to the lowermost point[4]. The rest of the vertices form what is called a monotone chain, which is characterized by the fact that traversing the polygon from the top, every vertex will be lower than the previous one. If a monotone mountain is put with its base extending horizontally, it resembles a mountain range, hence its name.

If the inside angle between two edges of the polygon is less than π, the vertex is *convex*. If the triangle described by a convex vertex and its two neighbors contain no other vertices, the triangle is an *ear* of the polygon. Any convex vertex of the monotone chain is the tip of an ear, with the possible exception of the vertices of the base. Cutting such an ear off from a monotone mountain will always result in a smaller monotone mountain. This leads to a simple linear procedure for triangulating such polygons: Start with a list of all these convex vertices, and run through the list, cutting off each convex vertex and creating a triangle. For each vertex that is being cut, if its neighbors were not already convex, see if they have now become convex, and if either one has, add it to the end of the list. Once you have reached the end of the list, you are done. Needless to say, the implementation contains this algorithm, but this is straightforward, and will not be described in further detail. The algorithm is summarized in table 1.

Table 1: Monotone Mountain Triangulation

```
Initialize an empty list
Add all convex vertices, excluding the endpoints of the base, to the list
While list is not empty
    Cut off the corresponding ear of the monotone mountain
    Delete the vertex from the list
    For the two neighbors of the vertex
        if the neighboring vertex is not an endpoint of the base, and was made
        convex by cutting off the ear, then add this neighbor to the list
```

Seeing how trapezoidations can be linearly triangulated, one can understand the search for fast trapezoidation algorithms that started in 1984. There are many ways of

[4] This depends on our choice of direction of the rays. With vertical rays the base would extend from the leftmost to the rightmost vertex.

computing the trapezoidation. All of them build on adding one small subset of the edges at a time into the current set of regions, creating more regions as one goes along. The subset can be a sequence of edges, or just one edge, depending on the algorithm. Further, they rely on two different phases, one to find the place to start adding the current part into, the second one to insert the subset and create new regions. Time bounds range from $O(n \log n)$ to $O(n)$ depending on the choice of how to do these two phases, and if a part is simply an edge or a collection of edges. While most of these are deterministic algorithms, some, like Seidel's, are randomized and the time bound is an expected time bound.

3 Seidel's Trapezoidation Algorithm

The choice on which algorithm to implement fell on Seidel's algorithm for one reason. It is the only one of the near-linear algorithms that is feasible to program. The implementation originally started as a class project in an Algorithms class together with Alex Burst. We successfully implemented a simple version of this algorithm in Mathematica. It managed to triangulate simple polygons, and could output graphics and movies of the trapezoidation. This version never achieved near-linear running time. For this project, I wanted to improve on the time used by the algorithm, and further generalize and optimize it. I also wanted to compare it to other existing algorithms.

Seidel's algorithm is a randomized way of computing a trapezoidation. The complexity analysis is based upon probabilistic expectancies about for instance the depth of a binary lookup tree. In order to use expectancies, we need to have some random variables. Nothing is assumed about the input, but it is randomized in the sense that every order of adding the edges is equally likely. At any stage it will then be equally likely for the next edge to be added to be any one of the remaining edges. In this way one can use expectancies to estimate the running time. This will not estimate the worst-case behavior, but it is unlikely that the actual running time will be significantly worse than the expected running time.

Any trapezoidation routine starts with an empty plane, and adds in subset by subset of the polygon. In Seidel's algorithm each subset consists of one edge only: The algorithm first checks to see if the endpoints of the edge have been added to the trapezoidation already. Those that have not been added yet are added one by one, and divide the regions they are added into in two by means of a horizontal ray extending in each direction from the point. Next the edge is added between the two points. This is done by starting from the top point, and moving down one region at a time until the bottom point is reached. For each region traversed in this manner, the region is divided in two by means of the new edge. Whenever this causes two regions on top of each other to have the same left and right hand boundary, these two regions are merged into one. Each of the regions created in this manner will be a trapezoid. Note that in doing this, we never need to explicitly calculate the intersections of edges and rays, it suffices for each trapezoid to know which edges and rays it is bounded by.

Since we treat points lexicographically, no two points will be considered to have the same height, i.e. y-coordinate, and therefore each trapezoid will have only one upper and one lower bounding vertex, and no more than two neighbors above or below. Further this means that no edge will be horizontal, and the meaning of "left of" an edge will be well

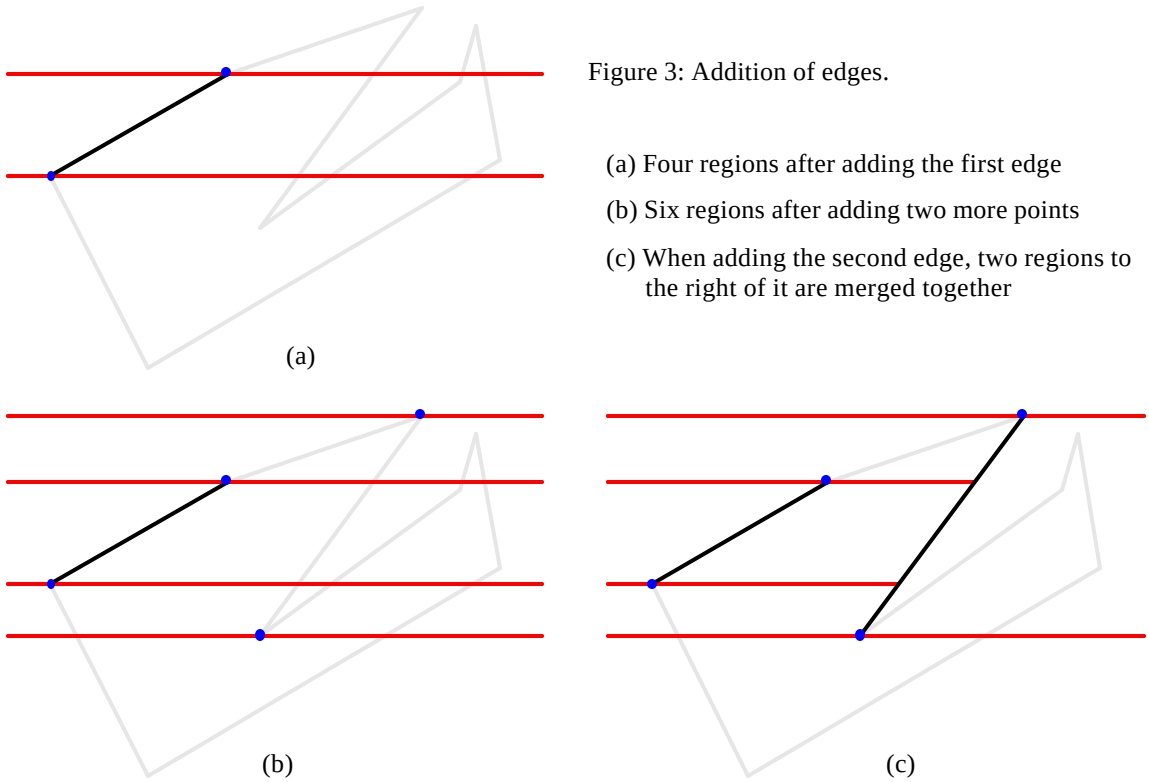


Figure 3: Addition of edges.

(a) Four regions after adding the first edge

(b) Six regions after adding two more points

(c) When adding the second edge, two regions to the right of it are merged together

defined. When the algorithm has split one region during the addition of an edge, it needs to find the correct region below to move into. If there is only one region, this is trivial, if there are two, then the correct one can be found by checking whether the endpoint of the edge being added is to the right or left of the edge that separates the two regions.

The algorithm also needs to efficiently be able to tell where to place new points in the current trapezoidation. One way of doing this is by creating a tree lookup structure for the trapezoids. The tree starts with only a root representing the original empty plane, and each time a region is split in two, the corresponding leaf in the tree is changed into a node with two children. The node represents the edge or the vertex that split the trapezoid, the two children represent the two resulting trapezoids. In this way each leaf in the tree corresponds to a trapezoid of the trapezoidation and each node represents either a vertex or an edge. By querying whether the point we are adding is above or below a vertex in the tree, or to which side of an edge it is, we can move down the tree to find the correct trapezoid containing the point. Lemma 1 shows the expected query time of this tree structure.

Lemma 1: If edges are added in a random order, then after i edges have been added the expected depth of the tree will be $O(\log i)$.

Proof (sketch; see [Seidel]): The tree is built by splitting the leaves representing the trapezoids into which a vertex or an edge is added. By randomizing the order in which the edges are added, at any step, the vertex to be added next resides in any trapezoid with equal probability, and any leaf has an equal probability of being split. Any new trapezoid created by the introduction of a vertex thus has an equal probability of being put below any of the old leaves, so trapezoids will be distributed across the leaves of the tree with

equal probability. An edge splits the trapezoids between its two end points, and since these are distributed across the tree, so will the new trapezoids be. In total this ensures us that all the trapezoids will be randomly distributed across the tree. When a tree is generated by randomly selecting a leaf and splitting it, after j splittings, its expected depth will be $O(\log j)$. Each point added will create one new trapezoid, and due to merging, each edge will only generate one new trapezoid as well. After the addition of i edges, the number of trapezoids will be $O(3i)$, and the expected depth of the tree is then $O(\log 3i) = O(\log i)$.

Once both the points of the edge have been added to the trapezoidation, the edge should be added too. Lemma 2 shows that the expected number of trapezoidal boundaries an edge will have to traverse is $O(1)$, which is at most a constant.

Lemma 2: If edges are added in a random order, then after i edges have been added the expected number of horizontal rays intersecting an edge not added is at most 4.

Proof: Each point added will make two horizontal rays. Thus after j edges have been added to the trapezoidation, there are at most $2j$ points, and at most $4j$ horizontal rays in the trapezoidation, each of which may abut upon one other edge. Because of the random ordering of the edges, any ray is equally likely to hit the k-th edge added as the l-th edge added, for some k and $l \leq j$, and the expected number of rays hitting any one edge is at most $4j / j = 4$, for any j. The number of horizontal rays that will intersect an unadded edge after i random edges have been added, is the same as the number of rays that would abut upon this edge after addition if this edge were added next, i.e. after $j = i + 1$ edges had been added. By the analysis above this number is expected to be at most 4.

This means that the number of horizontal rays an edge has to cross when it is being added is $O(1)$, and the number of trapezoids that have to be split is one more. In total this leads to an algorithm which has an expected running time of $O(n \log n)$; expected constant time for each of n edges, and expected logarithmic time for each of n point locations. See [Seidel] for further details.

By using the fact that in a polygon all the edges are connected, this time bound can be further improved upon. The algorithm can be divided into $\log^* n$ phases, where each phase first adds in some edges, and afterwards the remaining points are located in the trapezoidation by traversing edges as described above, but this time without introducing new regions. Finding the correct trapezoids for new points will now be cheaper, as we no longer have to start from the root of the tree. We get the following algorithm:

1. Generate a random ordering of the edges
2. For each of $\log^* n$ phases do
    - 2.1 For each edge in the current phase do
        - 2.1.1 If endpoints are not added, find them through the tree and add them
        - 2.1.2 Add the edge between the two endpoints
    - 2.2 If there are more points to add then
        - Locate the remaining points in the tree by tracing edges through trapezoids

Assume that step 1 can be done linearly (See [Seidel] for a discussion of this). Step 2.2 will expectedly take $O(n)$ time; the location of at least one point is known in the tree, and tracing an edge towards the next point is $O(1)$ for each edge, by lemma 2. Step 2.1.2 will take expected constant time per edge, for total expected time $O(n)$, also by lemma 2. In section 7 it is proved that by scaling the size of each phase correctly, the total expected time for all tree queries during a phase becomes linear in n, and thus step 2.1.1 will also take expected time $O(n)$. Thus the expected overall running time of step 2, and the algorithm in general is $O(n \log^* n)$.

4 Implementation Issues

The above algorithm has been successfully implemented. There are many ways to do so, and the one described here could be optimized, but is kept general to allow for the extensions mentioned in section 5. Each point, edge and region is labeled with a number. There are n points and edges, and initially only 1 region - the empty plane. To avoid searching trough the memory structures, many structures need lookup functions in both directions. Locating the trapezoids next to a point needs to be done in constant time without searching, so there is a structure that remembers where points are in the trapezoidation. Likewise locating the points next to a trapezoid must also be done in constant time, so there is a structure going in the opposite direction. Constant lookup time for all such arrays is assumed.

4.1 Memory Structures

Vertices: For each point in the original polygon we need to keep its x and y coordinates around.
EdgeLow and *EdgeHigh*: Edge i will generally go from vertex i to vertex $i + 1$, so there is no particular need to store the upper and lower point of each edge. The algorithm often needs to know about the high or low points of the edges, so having this in a nice format speeds things up. This will also allow us to nicely generalize the algorithm later.
PointsEdges: Point i will generally have edge i-1 incoming and edge i outgoing, so just as for *EdgeLow* and *EdgeHigh* there is initially not much need for having a complete structure for this. All PointsEdges are defaulted to i-1 and i, and only those that differ from the general rule are actually stored. Again, this structure will allow for some nice generalizations on the algorithm.
RegionCounter: An integer counting how many regions there are. Initialized to 1, and incremented each time a new region is added.
Regions: Each region needs to know its boundaries. It knows its neighbors above and below, and the edges on each side. These structures are updated frequently as more points and edges are added. Each time an edge is being added, it will split the current trapezoid, and then choose a region below it to proceed into. In addition each region must know which leaf it is in the tree, so that anything that splits this region also can update the tree. The first region is bounded by ∞ and $-\infty$ to the sides, and has no neighbors above or below. It points to node 1 in the tree.

RegionsAbovePoints: One needs to be able to tell between which trapezoids the edges and points are located. The way this has been implemented is by having each point remember the trapezoids above it. There might be 0, 1 or 2 edges going up from a point, so 1, 2 or 3 trapezoids must be stored. To find the trapezoids to either side of an edge, one can use a combination of EdgeLow and RegionsAbovePoints.
NodeCounter: An integer that counts how many nodes and leaves there are in the tree. Initialized to 1 and incremented by 2 as nodes are split.
Tree: Each node in the tree is either a vertex; an edge; or a leaf representing a trapezoid. The nodes must know their type, the edge- leaf- or region number, as well as their two (or none) children. The initial tree has just one leaf pointing to the initial region.
PointsAdded: An array of length *n* all initialized to false. Each time a point is added to the trapezoidation its value is changed to true. This ensures us that no point will be added twice, once for each of its two edges.
PointsTreePosition: After the completion of each phase points are located in the trapezoidation, so they can easily be found in the tree. The leaves corresponding to the region found are stored in this structure.
TrapezoidPosition: A trapezoid can be either inside or outside the polygon. These are not found until after all the edges have been added.
SegmentOrder: A list that contains the order in which to add the edges. This list is a random permutation of the integers from 1 to *n*.

4.2 Edge addition

The first step to perform is to create a random permutation of all the edges. This is done by an internal piece from Mathematica. Testing shows this to perform linearly. After the completion of each phase, during step 2.2, the remaining vertices are located in the trapezoidation by starting from the position of the previous vertex, and traversing trapezoids until the next location is found. These tasks are relatively simple to implement, and will not be discussed further.

Given an edge to add, the algorithm first checks *PointsAdded* to see if the end points of the edge have already been added to the trapezoidation. If either one has not, it is now added to the trapezoidation. If both should be added they should be added in random order, to satisfy the assumptions of lemma 1. When finding the correct trapezoid to add the vertex in, we start from the node in the tree dictated by *PointsTreePosition*. This node will probably no longer be a leaf, so the tree is traversed from that node down until a leaf is encountered. If an encountered node is a vertex, one takes the left child if the point to be added is higher, and if the node is an edge, one takes the left child if the point to be added is to the left of the edge. Once the correct leaf is found, the point is added in the region that the leaf points to. A new region is added which is a copy of the region where the point was, and the neighbors above and below of these two regions are updated. The leaf in the tree is transformed into a vertex node and given two children, the left one corresponding to the old region and the right to the new one. *RegionCounter* is incremented by 1 and *NodeCounter* by 2. This is summarized in table 2.

Once both endpoints have been added to the trapezoidation, the edge can be added between them. To simplify the programming, all edges are added from the top point and down. Each trapezoid encountered on the way is split in two, the regions and the tree are

Table 2: Point Addition

If point is not added already then
    Start from the location in the tree given by *PointsTreePosition*
    Traverse tree downwards until a leaf is found
    Change the leaf to a node representing the point, and give it two children
    Split the corresponding region in two, make top region's neighbor
     below be the lower region, and vice versa
    Make the new regions point to the new children in the tree, and vice versa
    Increase *RegionCounter* and *NodeCounter*

updated, the correct trapezoid below is found, and this is then split in the same way. By convention a region to the left of an edge becomes the left child of a node and vice versa. When an edge is being added, a trapezoid to be split may have one of 18 different configurations depending on the neighbors of the trapezoid. By querying on the neighbors of the trapezoid, the configuration could be found and the regions updated correctly. However, there is also a need to update the up and down pointers of each region, so one must consider the possible configurations of the trapezoids following below. There are 180 different combinations of trapezoids when adding an edge, so it becomes much easier to generalize and group similar configurations. As a result there are several procedures to do this. *SplitCurrent* takes the current region, splits it into two and updates the tree. *UpdateFirst* will be called after the first splitting just below the top point. Next *UpdateRAPs* is called which updates the *RegionsAbovePoints* for the points nearby. *FindNextRegion* then finds the next region below. After this region has been split with *SplitCurrent*, *UpdateMiddle* is called to update the neighbors of the new regions, then *UpdateRAPs* and *FindNextRegion* again. Once the last region has been split, *UpdateLast* is called to update the regions around the lower point. This is summarized in table 3.

Each of these functions work independently of the others, and each has a number of

Table 3: Edge Addition

Let *CurrentRegion* be the region immediately below the top point
SplitCurrent
UpdateFirst
UpdateRAPs
CurrentRegion = FindRegionBelow
While *CurrentRegion* is not the region immediately below the bottom point
    SplitCurrent
    UpdateMiddle
    UpdateRAPs
    CurrentRegion = FindRegionBelow
UpdateLast

queries on the neighbors and boundaries of the trapezoid to find the correct way to update the regions. Two of these will be described in some further detail.

UpdateFirst updates the pointers to neighboring regions just below the top point. If there is no other edge attached to the top point, this is easy, the one region above should now point down to the two new regions, and the two new regions should point up the old one. If there was an edge going up from the point, then the regions on either side should now get only one neighbor below each, and likewise for the two new regions. If there was an edge going down from the point, one of the new regions should now have no neighbor above, the other should have one, and the region above needs to point down to the correct regions. See figure 4.

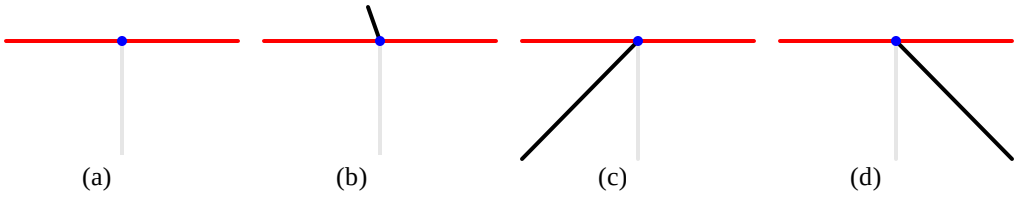


Figure 4: The 4 different possibilities for UpdateFirst when the grayed out edge is being added. (a) no other edge attached to the point, (b) an edge going up, (c) an edge going down to the left, and (d) and edge going down to the right. Each case requires the pointers of the trapezoids to be updated differently.

UpdateMiddle takes care of updating the regions that are neither just below the top point, nor just above the bottom point. In the current implementation, this procedure has been worked in as a part of *SplitCurrent*. There are too many possible configurations to describe them all, but one slight optimization over Seidel's original routine will be discussed: When edges are splitting a trapezoid in two, it sometimes happens that two regions lying on top of each other now have the same right and left boundaries, and in fact should be just one trapezoid. See figure 3 (c). Seidel proposes to go back to stitch such regions together after the edge has been added. In fact, every splitting of a trapezoid due to an edge, apart from the first trapezoid below the top point, will create two regions that should be merged. Whenever the edge is traversing a horizontal ray, the ray must be cut off on one side, and two regions merged. Recognizing this, one does not have to create a new region at all, it is enough to extend a region from above down to the current level. By doing this, a second run through the trapezoids can be avoided, and much overhead eliminated. When this is being done, one needs to take care to update the tree correctly. The current trapezoid will be split in two, and there must be two children of its node in the tree. One of these will now point to an already existing leaf, so strictly speaking the tree is no longer a proper tree as it may have multiple paths to a leaf. See figure 5 for an example of this.

Each one of these procedures called during an edge addition will update regions and the tree, and after the entire edge has been added, all of the following memory structures should have been changed or updated; *RegionCounter*, *NodeCounter*, *Regions*, *Tree*, *RegionsAbovePoints* and *PointsAdded*.

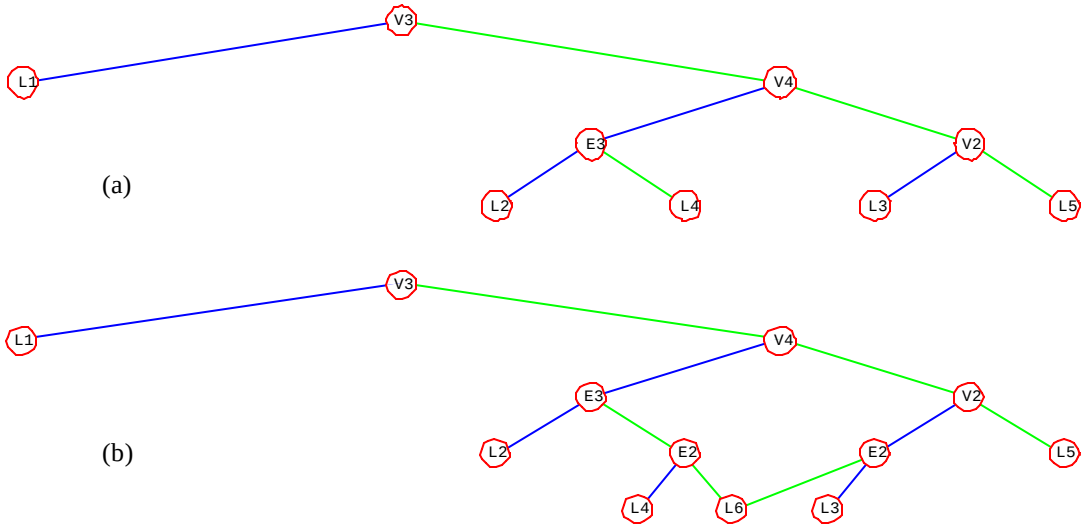
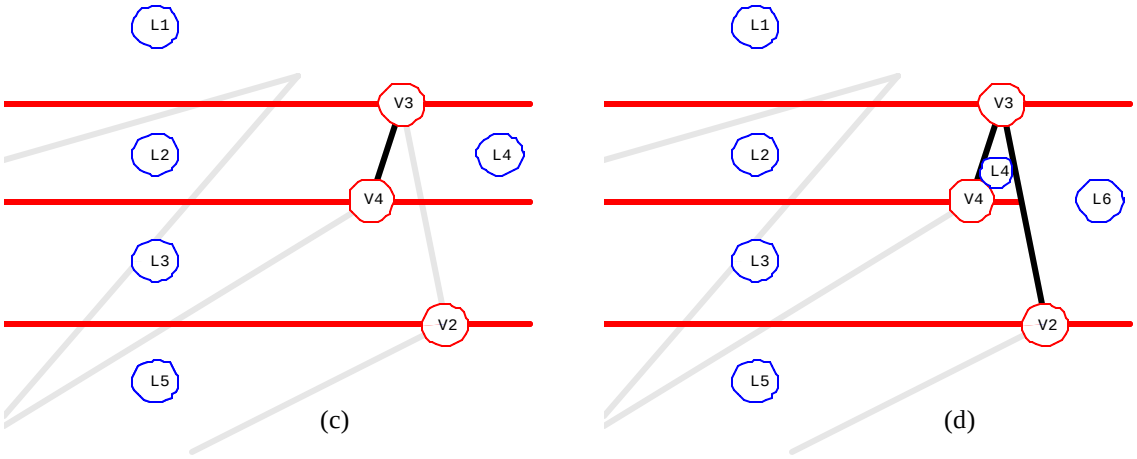


Figure 5: An example of the tree structure. 5(a) shows the tree after the edges of 5(c) have been added, and 5(b) that after the edges of 5(d) have been added. For vertices, blue means above, and green below, for edges, blue is left, and green is right. Note how adding edge E2 creates two branches to region L6.



Once all the edges have been added the rest is relatively easy. By always putting the new region after a point addition below the old region, we can be assured that the topmost region is still number 1, and outside the polygon. Starting from this region, and visiting all its neighbors and their neighbors recursively, we can check off all the trapezoids outside the polygon. No trapezoid from the outside will have a neighbor inside. Now considering only the inside trapezoids, we can find those that have boundary points on opposite sides, and draw diagonals between these. As previously mentioned the polygon will now be partitioned into monotone mountains, which can easily be triangulated.

5 Generalizations on the Routine

The routine described above takes as input a simple polygon and returns a triangulation of the polygon. There are several ways the existing routine can be modified to be more general or perform better.

5.1 Simple Generalizations

The algorithm already uses a tree structure to locate points fast in the trapezoidation. Once the polygon is finished, and all trapezoids marked as inside or outside, this tree structure can be used to tell, in expected logarithmic time, whether a point is inside or outside the polygon, and which trapezoid it is in. Seidel mentions this in his paper, and this was fairly easy to implement. The routine now returns a list representing the tree structure, and another procedure can use this structure to efficiently test whether new points are inside the polygon or not.

More than just one polygon at a time can be triangulated. The only place where connectivity of the edges is assumed, is in the fast point location at the end of each phase, step 2.2. If we allow for several non-intersecting polygons, these too can be triangulated. The first point of each piece will have to be found through the tree only, as we can no longer use the connectivity to traverse the trapezoids. Seidel shows how a trapezoidation of several connected pieces can be computed in expected time $O(n \log^* n + k \log n)$, where k is the number of such pieces. This too was implemented without too much change in the initial program. In implementing this, the general structure of EdgeHigh, EdgeLow and PointsEdges comes in handy, as it now no longer is so certain that edge i connects vertex i and i+1. The last vertex of polygon 1 will now connect to vertex number 1 instead of the next vertex in the input sequence, which would be the first vertex of polygon number 2.

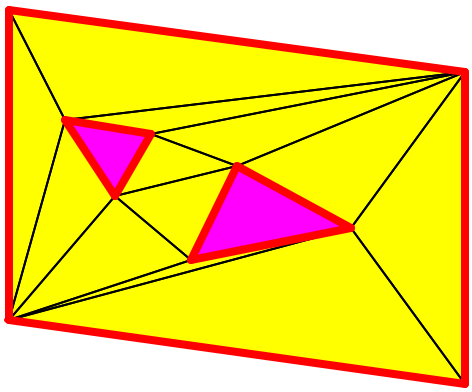


Figure 6: Triangulation of multiple polygons with varying triangle depth. Yellow is depth 1, and magenta is depth 2.

Given several polygons, some of which may be inside others, there can now be different depths to different triangles. See figure 6. The depth is defined as the minimum number of edges one has to cross to get to that triangle, starting from infinity. Making the depth of each triangle part of the output function is easy too: Modify *TrapezoidPosition*, which used to tell whether trapezoids were inside or not, to *TrapezoidDepth*. Initialize all of these to -1. Create two new empty lists, *ThisDepth* and *NextDepth*. We know region 1 is outside, so we put it in *ThisDepth* and set the current depth to be 0. Now recursively mark all of the neighbors of the regions in *ThisDepth* with the current depth. Each time an edge different than $\pm\infty$ is encountered, cross it by using *EdgeLow* and *RegionsAbovePoints*, and if the trapezoid on the other side has depth -1, add it to *NextDepth*. Once this recursion is completed, copy *NextDepth* into *ThisDepth*,

empty *NextDepth*, and start over. This is repeated until no regions were added to *NextDepth*, and summarized in table 4. When it is all done, each region will have been visited once, and given its appropriate depth. A simple analysis shows that this can be done in time linear in the number of trapezoids, which is linear in *n*.

Table 4: *TrapezoidDepth*

```
Initialize array of depths of regions to -1
Let CurrentDepth be 0
Create two empty lists, ThisDepth and NextDepth
Add region 1 to ThisDepth
Repeat until no regions were added to NextDepth
    Recursively visit all the neighbors of all the regions in ThisDepth
        Give each visited region depth CurrentDepth
        Find the region on the other side of both edges of the region
            If this region currently has depth -1, add it to NextDepth
    Let ThisDepth = NextDepth, and empty NextDepth
    Increment CurrentDepth by 1
```

5.2 Intersections

So far all that has been discussed has been simple, non-intersecting polygons. The world is often more complex, and the question becomes whether it is possible to use this algorithm to triangulate self-intersecting polygons. The answer is that it is possible, but some new memory structures are needed. The general extended algorithm will be presented first, then the new needed structures.

A single trapezoid can only have one edge as its boundary on either side. This means that we will have to treat intersections as points bounding the trapezoids, or a trapezoid might extend across an intersection and have two boundary edges. Let *l* be the number of intersections between edges, and *m* the number of points plus the number of intersections; $m = n + l$. Running time and memory usage of the algorithm will now be a function of *m* instead of *n*. Note that *l* is bounded by $1/2\ n^2$.

The extension of the algorithm is still fairly straightforward. When traversing trapezoids during the addition of an edge, we can no longer choose the trapezoid below based simply on a "left of" query with the lower point of the edge being added, we need to first check if one of the two edges of the trapezoid should have been intersected. Sometimes we will choose a region below, but sometimes we need to intersect an edge before we reach that region.

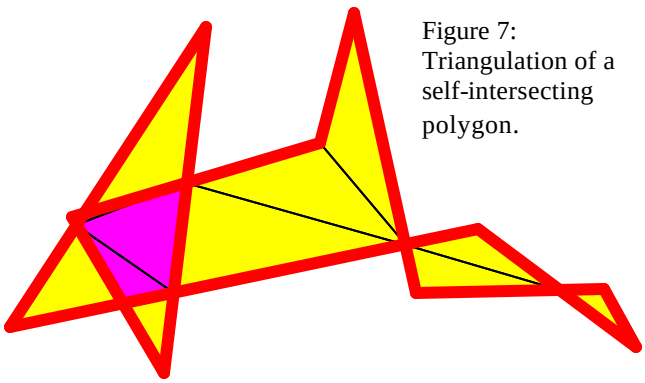
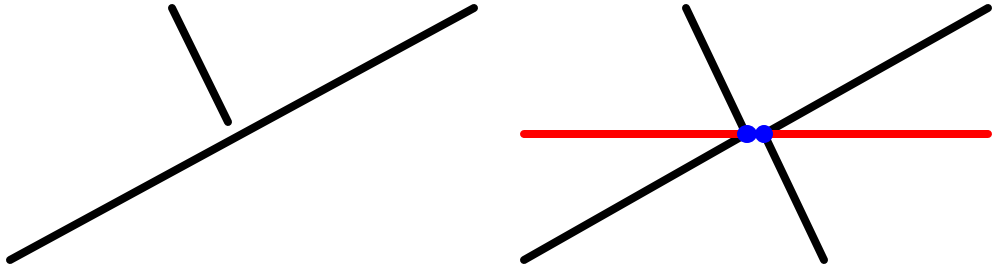


Figure 7: Triangulation of a self-intersecting polygon.

Although this conceptually is relatively simple, we now have to use divisions for the comparisons, and at this point it becomes very important to watch for roundoff errors. If we find that we should really be on the other side of an edge, we need to intersect that edge. The intersection point is found, and two new vertices added. The reason for adding two vertices is that when splitting the edge we intersect in two, we obtain four pieces of edges all ending at the same point. If we add two vertices, each vertex will have only two edges, and it is possible to traverse a polygon by edges and vertices only, and the rest of the routine can remain relatively unchanged.



(a) (b)

Figure 8: Intersection of edges: (a) just before intersection and (b) just after intersection. Two new points have been introduced, and regions on both sides have been split. Note that the two points have the exact same coordinates.

When finding an intersection point, two new vertices are added. One in the trapezoid we were just in, the other one in some trapezoid on the other side of the edge. There may be more than one trapezoid on the other side of the edge, consider crossing the rightmost edge of figure 3(c) from the right. It then becomes necessary to find the correct trapezoid, and since we want the time taken for edge addition to be constant, this ought to be found in constant time too. We only know the regions above the lower point of the edge, and we need to start searching for the correct trapezoid from there. Note that lemma 2 no longer holds; points have been introduced at intersection points, so the number of points does not have to be linear in the number of edges added. In section 6 it is proven that a similar result as that of lemma 2 holds for intersecting polygons too. This will allows us to expect the number of rays between the two end points of the edge to be constant, and therefore also the number of trapezoids we need to search through. Thus we can find the correct trapezoid in expected constant time, by crossing only an expected constant number of rays.

Before intersecting there were just two edges, and there are now four, so two of these edges will be new ones, and it is necessary to update the regions that have these edges as part of their boundary to point to the new edge. Of the same reasons as above, this too can be done in expected constant time for each edge. Once we have completed the updating, and moved to the opposite side of the intersected edge, we can continue traversing trapezoids downwards towards the bottom point.

If we are to be entirely general, we should also allow for collinear, overlapping edges and multiple vertices or intersections at the same location. The existing framework is not sufficient for allowing this. The problem arises when we have to find the correct way to the endpoint, when the nearby vertices all have the same coordinates. Consider the problem of several collinear and overlapping edges. In order to get the depth correct, we have to add all of them into the trapezoidation. This requires us to have trapezoids between the edges, even though these trapezoids have zero width. Let us say that we are currently adding an edge in the trapezoid between two overlapping and collinear edges. (Figure 9.) Let us further assume that both of these extend further down than the edge we are adding, and the current trapezoid extends down below the end point of the current edge too. In order to make sure all the trapezoids have the correct neighbors, we need to add the current edge all the way to its endpoint, which would be on the other side of one of the edges. Thus we need, in expected constant time, to be able to tell whether we should cross the left or the right edge. Note that in general there might be a function of n edges on either side, so we cannot rely on searches to find where the point is in expected constant time.

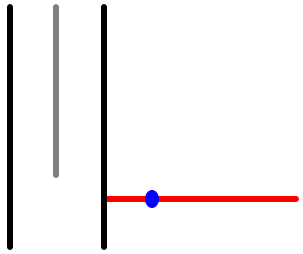


Figure 9: Overlapping edges. When adding a third edge between two overlapping edges, it becomes necessary to tell where the end point is by only looking locally. Note that the three edges and the point have the same x-coordinate.

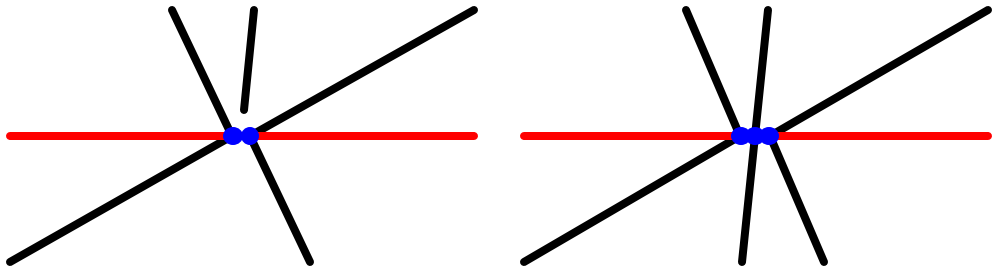
This problem can be solved by giving each point a unique position number. If two points have the same coordinates, then the position number will determine which one of them is considered to be higher. This becomes an extension of the lexicographic ordering of points to include points at the same location. When adding a new vertex with the same coordinates as an existing vertex, if the position number is lower, we will add it above, and if the position number is greater, it will be added below. Deciding whether to add the point in figure 9 to the left or to the right of the two existing edges with the same x coordinate can be done by comparing the position number of the point to that of the endpoints of the edges. By assigning position numbers randomly to the vertices, we can be assured that it is equally likely that the this point will be added to either side of either edge. In practice position numbers are given by creating a random permutation of the n vertices, in the same way edges are randomly ordered. In the current implementation, these two random lists are in fact the same, which can have the unfortunate consequence of increasing the time bound for some very specialized input.

In the situation in figure 9, since the point has been added to the right, it would have to have a greater position number than both the edges, and we would know that we would have to intersect the right edge to get to it. This also means that overlapping edges are treated in just the same way as simple intersections are treated, and we need not worry about special cases in the running time analysis.

We have to take care that when we intersect edges, the two new points introduced are given correct position numbers. If the intersection happens at the same location as an already existing vertex, the new vertices should be given position numbers that reflect their relative position to the edge that is intersected. In the example above two new vertices would be introduced in order to cross the rightmost edge. The left vertex would be given the number of the lower vertex of the intersected edge, and the right vertex the

same number as the end point of the edge we are adding. This ensures that we also later can find our way by looking around us only locally.

When an edge being added is going over the location of an already added vertex, position numbers are again used to determine whether the edge should go to the right or left of the vertex. If instead of a single point, we hit an intersection straight on during the addition, it would be extra work to intersect both of the two edges, one by one, in order to get across to the other side. By squeezing the edge we are currently adding between the two existing vertices, we do not need to intersect either edge, we can get away with a lot less overhead. Thus the number of points introduced per intersection is not a constant, but it will never be more than two for crossing a single edge. See figure 10.



(a) (b)

Figure 10: Multiple intersection of edges: (a) just before intersection and (b) just after intersection. Only one new point has been introduced. Note that all three points have the exact same coordinates.

Note that the choice of introducing two new points at an intersection now pays off. Since no point will have more than two incoming edges from above, *RegionsAbovePoints* will still have at most 3 regions stored, and we can still look at all of them in constant time without worrying about how long time it will take to search through this memory structure.

Two new memory structures are introduced to handle intersections:
PositionNumber: As explained, a random number for each vertex to determine where it goes relative to other vertices with the same coordinates.
OriginalHigh and *OriginalLow*: If there are many points at the same location, some edges will now go between two points with the exact same coordinates. It then becomes impossible to perform "left of" queries on the edge, so we need to remember in which direction the edge is going. Remembering the points in the initial polygon the edge is lying between takes care of this. This also reduces the chance of build-up of roundoff errors. If we were to calculate new intersection points based on previous intersection points instead of on the original polygon, the previous intersection points would have an error associated with them, and the new intersection we calculate might have an even larger error.

Once we have handled intersections, we are no longer in need of polygons to create triangles deep inside a structure. Given three intersecting lines, there will be a triangle between them, and a triangulation of lines as well as polygons makes sense. It is now trivial to extend the algorithm to take lines as valid input as well. Normally winding numbers, a function of the orientation of all the polygons containing it, determines the depth of a triangle. This will no longer give the correct answer as it fails to take lines into

the equation. The way depth is implemented now becomes essential for getting a good answer.

The issue of exactly what a triangulation of intersecting polygons means now becomes important. There is no agreed upon definition of this in the literature, and many different definitions can be made. Regardless of how the triangulation is defined, the algorithm described above will compute the unique and well-defined trapezoidation, and from this, with only minor modifications to the routine, almost any kind of triangulation can be obtained. For the purpose of my implementation, I have chosen to define a triangulation in terms of the depth of a simple polygonal region. Each region bounded by edges in the original polygons will be triangulated as if it were a simple polygon, and each triangle resulting from this will have the same depth as the region itself.

The implemented algorithm has been successfully enhanced to take very general input. Polygons are specified in terms of a list of vertices. It is always assumed that the last point is connected to the first point, if these two are the same, the last point is deleted. The orientation of a polygon is irrelevant, and it can have any kind of degeneracies such as intersections, collinear and overlapping edges and multiple vertices at the same location. Lines are specified as polygons of lengh two. The algorithm can take any combination of these, each specified as one element in a list, with any number of degeneracies between them. The output of the routine is a set of triangles, each one with its own depth, the three vertices of the triangles being either vertices from the input, or points where edges intersect.

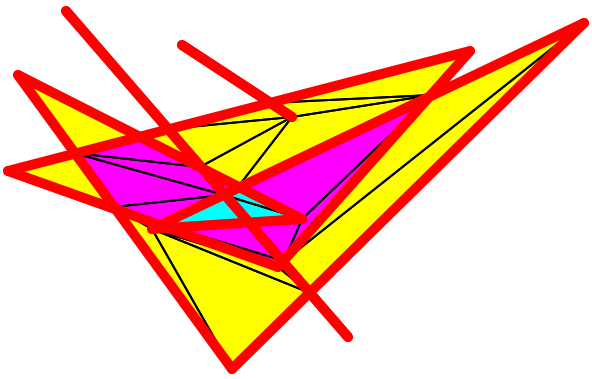


Figure 11: A general triangulation of multiple polygons and lines.

6 Complexity Analysis

Most of the memory structures are linear in the input size, since there is one slot of constant size for each point or edge. The number of trapezoids is linear in the number of points, so the *Regions* structure is of linear size too. The tree has a leaf for each trapezoid, and is therefore linear in the number of such. Thus the memory needed for the implementation is strictly linear in the number of points. For simple polygons, memory use is $O(n)$, and for intersecting polygons it is $O(m)$.

For now, let us be concerned with the running time of a single simple polygon only. By lemma 2, the updating of a point during step 2.2 is expected to be constant. Start at a point which has a known location in the trapezoidation, update point by point of the polygon by tracing each edge from a point for which the location is known to the new point. Each edge is expected to hit at most a constant number of trapezoids, so if the initial point in the polygon is known, this takes expected time $O(n)$ to do for all the points.

Given log* phases, each point will have its location updated at most log* times, and the point updating for all points takes at most $O(n \log^* n)$ time. The addition of each edge takes expected constant time plus the time taken to find the new location of the points through the tree, given their last known location. All we need to show is that given a correct division into the log* phases, the total expected time of the tree search is linear during each phase. If we can show this, edge addition for all edges will take at most expected $O(n \log^* n)$ time, which will be the overall expected running time of the algorithm.

Lemma 3: If edges are added in a random order, then after i edges have been added the expected query time to find a point in the tree will be $O(\log (i / j))$, where j is the last time the point update of step 2.2 was performed.

Proof: All nodes added after edge j in the tree are expected to be equally distributed between the leaves of the old tree from stage j. The number of regions at stage i is linear in i, so an expected $O(i - j)$ regions have been added across the j old leaves. This tells us that an expected $O((i - j) / j) = O(i / j)$ nodes have been added to each one of the old leaves, and the query time to find a new leaf starting from one of the old ones will expectedly be $O(\log (i / j))$ by lemma 1.

Let $\log^q$ denote the q-th iterated logarithm, i.e. $\log^0 a = a$, $\log^1 a = \log a$, $\log^2 a = \log \log a$, and so on. Log* n is then the smallest integer q such that $\log^q n \leq 1$.[5] During phase p, let us add edges from the randomized sequence starting from $n / \log^{p-1} n$, and ending at $n / \log^p n$. Phase 1 will then start at edge number $n / \log^0 n = n / n = 1$, and the last phase ($p = \log^* n$) will end at edge number $n / \log^{\log^* n} n \geq n$, since $\log^{\log^* n} n \leq 1$. In the implementation, the last edge added of each phase is the ceiling of $n / \log^p n$, with the last phase ending at the n-th edge. When adding the i-th edge during phase p, the tree search will be of expected time $O(\log (i / j))$ where j now equals $n / \log^{p-1} n$. $O(\log (i / j)) = O(\log (i / (n / \log^{p-1} n))) = O(\log (\log^{p-1} n * i / n))$. Since $i \leq n$, each point location during phase p will expectedly cost $O(\log \log^{p-1} n) = O(\log^p n)$. Since phase p ends at randomized segment number $n / \log^p n$, the total expected cost of point locations during phase k will be at most $O((n / \log^p n) * \log^p n) = O(n)$.

The cost of adding an edge is expected to be constant by lemma 2, so the addition of the edges during a phase once the locations of the points are found will be sub-linear. The total work done during any phase is then expected to take at most linear time. The total expected time of adding the edges will then be $O(n \log^* n)$. As shown above this is also the time taken to update the positions of the points. For a more detailed proof, see [Seidel].

Seidel also proves that the expected time for several non-intersecting simple polygons is $O(n \log^* n + k \log n)$, where n now refers to the total vertex count, and k is the number of such polygons,. The increase in the bound comes from locating the first point in a polygon. If polygon i has not had any of its edges added into the trapezoidation yet, finding the first point of polygon i during the update after each phase will take expected logarithmic time in the number of edges added already, by lemma 1. The

[5] Some authors define log* n to be the largest integer q such that $\log^q n \geq 1$.

number of edges added may be of $O(n)$ for an expected search time of $O(\log n)$, and there are k polygons. This can easily be decreased by noting that at least one of the polygons must always have an edge added during the first phase, and thus there are only k-1 polygons left that must be looked up in the tree structure for an expected bound of this to be $O((k\text{-}1) \log n)$. A further decrease can be achieved by requiring that the k first edges added be from k different polygons. As long as these k first edges are added in a random order, all possible permutations of the tree after the first k edges will still be equally likely, and lemma 1 still holds. Now the first point in each polygon can be found through the tree faster since the tree has not yet grown too big when these points are being added. The total expected time of inserting polygon i, as one of the first k edges, into the trapezoidation then becomes $O(\log i)$ for each of k polygons. $O(\log i) = O(\log k)$, so the expected sum of this over all the k polygons becomes $O(k \log k)$. The result is that the expected running time with k simple polygons is $O(n \log^* n + k \log k)$. Asano et al. proved in 1984 that the triangulation of simple polygons with holes is $\Omega(n \log n)$. This result still holds, but if the number of polygons is not $O(n)$, it can be reduced as shown here.

It still remains to show that the time bounds also holds when the polygons are allowed to intersect. None of the extra vertices introduced at intersection points have to be searched for through the tree, as they are added in as soon as they are found. The update after each phase thus needs to update the same number of points. What changes is the number of vertices in the trapezoidation, and thus also the number of horizontal rays and the number of trapezoids, so lemma 2 no longer holds. With l intersections, edge addition will no longer be constant with respect to the number of edges added, but rather to the number of points in the current trapezoidation, including intersection points.

After the addition of the i-th edge, due to the random ordering of the edges, the expected number of intersections in the current trapezoid will be $O(i / n * l)$, where l is the total number of intersections. At any point in the algorithm there will then be an expected number of $O(i / n + i / n * l)$ points added, and the expected number of trapezoids split by an edge increases from $O(1)$ to $O(1 + 1)$. This would lead to an overall algorithm of expected running time $O(nl \log^* n + k \log k)$, far from the near-linear we started out with.

Notice however, what would happen if we knew the number of intersections ahead of time. We would treat each intersection as a point, and have a total of $m = n + l$ points. The intersection points would not have to be updated at the end of each phase, so we can still assume the same connectivity of the polygons at this stage as before. The expected running time would be $O(m \log^* m + k \log k)$, which is easy to simulate. Before the end of each phase, we calculate the expected m based on the number of intersections seen so far, and if the new expected value is greater than the previous value, we extend the length of the phase correspondingly. During the first phase, our expected value for m will be n. We will be adding in edge by edge, but in order to determine when to end a phase, we will not look at how many edges have been added, instead at the number of edges added plus the number of intersections found. Once all the edges have been added we are done, even if we have not completed the last phase.

The extension of lemma 1 is straightforward, the depth of the tree will now be a function of the number of edges in the trapezoidation, not of the number of edges added

from the original polygon. We also need to show that the equivalent of lemma 2 still holds:

Lemma 4: If edges are added in a random order, if there are i edges in the trapezoidation, the expected depth of the tree will be $O(\log i)$.

Proof: As the proof of lemma 1.

Lemma 5: If edges are added in a random order, if there are i edges in the trapezoidation, the expected number of rays intersecting an edge not added is at most $4(j+1)$, where j is its number of intersections with other edges.

Proof: As the proof of lemma 2. If there are i edges in the trapezoidation, there are at most $2i$ points, and at most $4i$ horizontal rays, each of which may abut upon one other edge. Any ray is equally likely to abut upon any edge, and the expected number of rays abutting upon an edge is at most 4. The number of horizontal rays that an unadded edge will intersect is the same as the number of rays that would abut upon it if it were added next. If it were added, it would split into $j+1$ edges due to its intersections, and the expected number of rays hitting these $j+1$ edges would be at most $4(j+1)$.

By calculating the expected value of m we can therefore reduce the expected running time of the algorithm. We change from phase to phase depending on how many edges there are in the trapezoidation, not depending on how many edges we have added from the original polygon. We count from 1 to m, incrementing each time we start adding a new edge, and each time we intersect an edge. The algorithm from section 3 is generalized here, now with n replaced by $o = E(m)$. The actual number of phases done will not be known until the last phase is completed.

1. Generate a random ordering of the edges and *PositionNumber*
2. For $i = 1$ to $\log^* o$ phases do
    - 2.1 While there are fewer than $o / \log^i o$ edges in the trapezoidation
      Add the next edge from the randomized ordering by
        - 2.1.1 If end points are not added, find them in the tree and add them
        - 2.1.2 Add the edge between the two end points
    - 2.2 Recalculate o. If o increased then goto 2.1
    - 2.3 If there are more edges to add then
      Locate the remaining points in the tree by tracing edges through trapezoids

Step 1 is unchanged. Step 2.3 will take expected time $O(j+1)$ for each edge, for an expected running time of O(number of edges and intersections not in the trapezoidation) $= O(m)$. Step 2.1.2 will take expected time $O(j+1)$ time per edge, and add $j+1$ edges to the trapezoidation, for a total expected time of $O(m)$. Step 2.1.1 takes at most expected time $O(\log (r/s))$ by lemma 3, where r is the number of edges in the trapezoidation, and s is the last time step 2.2 was performed. s is at least $o / \log^{i-1} o$, and r at most o, for at most expected time $O(\log (o / (o / \log^{i-1} o)) = O(\log (\log^{i-1} o)) = O(\log^i o)$ for each point. Phase i ends at edge $O(o / \log^i o)$, so the expected running time is at most $O((o / \log^i o) * \log^i o)$

= $O(o)$. We are dealing with the expected running time, so since $o = E(m)$ we can replace o by m , for an expected running time of step 2.1.1 of at most $O(m)$. o can only increase if step 2.1 adds at least one edge, so step 2.2 will be performed at most n times. Step 2.1 and 2.3 are performed log* o times, and since $o = E(m)$, this leads to an overall expected running time of $O(m \log^* m)$.

The expected value of m can easily be calculated through calculating the expected value of l. For each possible edge intersection among the first i edges, the chance of finding an intersection there is equal that of finding an intersection between any other set of edges, again due to the random ordering of the edges. From this we expect the ratio of the number of such intersections found to the number of possible intersections to be constant. After i edges, there are a total of $1/2\ i(i\text{-}1)$ possible intersections; each edge can intersect any other edge already in the trapezoidation. If after i edges we have found j intersections, we expect the ratio $j / i(i\text{-}1)$ to equal the ratio after all the edges have been added, i.e. $l / n(n\text{-}1)$. Solving for l we expect that $l = j * n(n\text{-}1) / i(i\text{-}1)$, and we have a formula for o. The standard deviation for this estimator has not been found, and it might be possible to find some other way of estimating m that will be practically better.

It only remains to show that multiple polygons are not affected by possible intersections. Remember that the $k \log k$ part of the running time was got from summing $\log i$ over k polygons. But now there might be more than i points in the trapezoidation when we add the i-th edge. Up to $1/2\ i^2$ extra points might have been added, so now we need to sum over $\log (i + 1/2\ i^2)$ instead. For $i > 1$, $i + 1/2\ i^2 \le k^2$ since $i \le k$, and the sum of k terms of $\log k^2 = k \log k^2 = 2\ k \log k$. The expected cost of looking up the locations of the first k edges added from the tree is then $O(k \log k)$ as before.

It has then been shown that even for very general sets of intersecting polygons, it is possible to triangulate them in near-linear time in the number of points and intersections. The number of intersections may be quadratic in n, so worst-case behavior is $O(n^2 \log^* n)$. For most realistic cases both the number of intersections, and the number of different polygons is small when compared to the number of vertices, so this still runs in near-linear time in n. In the worst case the algorithm runs in near-linear time in m.

Although constantly checking for intersections and updating o does not decrease the asymptotic running time of the algorithm, it does decrease the actual running time. In order to avoid this, it is left as an option for the user to determine whether the procedure should be checking for intersections or not. This can improve running time in the cases where it is already known that the input is non-intersecting.

7 Mathematica implementation

The program has for the most part been implemented as described above, although in some places the description is somewhat simplified, or I have chosen more descriptive names of procedures. There are also parts of the generalized algorithm that have not been fully implemented yet. This section shows how some more specific implementation choices have been performed.

The output of the program is not a set of triangles. The actual output is an object of type *TriangulatedPolygon* that contains a number of different elements. By passing this object as an argument to different functions, the wanted output can be obtained. Any such

object contains; (a) the initial polygon, (b) the resulting set of vertices, including the intersection points, (c) the number of intersections, (d) the triangulation given as a set of triplets pointing to vertices of (b), (e) the depth of each triangle, (f) the tree structure for queries on the depth of points, and for debugging purposes (g) the order in which the edges were added and (h) timing data.

Various procedures operating on a *TriangulatedPolygon* object allows the user to obtain any of the above data structures, or to use them for other purposes. *PointDepth* takes a *TriangulatedPolygon* and a point, and determines the depth of the point. This can also be used to tell whether a point is inside the polygon or not, if the depth is 0, the point is outside. Several graphics procedures can take both polygons and *TriangulatedPolygon*s as input.

The user has a number of different options available to him. As mentioned earlier, it is possible to specify not to check for intersections, which will make the algorithm run faster, and will require a lot less computations on the way. If there are intersections anyway, the algorithm risks halting or running into infinite loops with no warning signs. If the algorithm does not return correctly, it is possible to abort it and call it again to obtain the order the edges were added in so the error can be tracked down. It is also possible to check for intersections, but to abort the algorithm if intersections are found, effectively making this a near-linear polygon intersection test. For testing purposes it is possible to specify the order in which the edges should be added. The floating-point error term can be defined, through experimentation I have found 10^{-10} to be a practical constant, so this is the default. A variety of different print statements are automated from within the routine. By turning various flags on, the algorithm will output step by step what it is doing. Some of these flags are for edge additions, point updates, intersects, or monotone mountain triangulation. In addition there are two flags which will cause the routine to output graphics of the procedure. *MovieQ* shows the current trapezoidation after each edge addition, after diagonals have been found and during the triangulation of monotone mountain. *TreeQ* shows a graphical version of the tree as it is being built. The result of this is movies showing step by step how the algorithm operates.

There is graphical output for a variety of different purposes. Showing the polygon, the trapezoids, the trapezoids and diagonals, or the entire triangulation are all possible. A graphics procedure that shows the relative depth of all the different triangles has been written. Tree structures can also be output graphically, both the lookup tree, and the tree-like structure of the neighbors of a region. Combinations can easily be obtained, such as the depth structure of a polygon along with a point, to visually check whether *PointDepth* returns the correct value. Automatically plotting the timing data of *TriangulatedPolygons* is also available. Most of the graphics contained in this paper was automatically generated using these routines.

All of these procedures have been implemented, but could still be improved upon. The implementation is not intended to be an industrial strength triangulation routine, but rather a tool to experiment with and to check real-time running times of the algorithm. This also holds for the rest of the routine, many parts could be further optimized to run faster, but the actual speed is not the main issue.

It should be noted that asymptotic running time for self-intersections is not $O(m \log^* m)$, but actually $O(n^2 \log^* n)$. The reason for this is that step 2.2 and 2.3 have still not been written for the general case. *FindNextRegion* is probably the most complex

part of the entire implementation[6], and since the algorithm was developed parallel to the implementation, it is not a modular part. Point updates through tracing trapezoids is very similar to actually adding an edge and finding the next region through *FindNextRegion*, so this is possible to implement without changing anything else in the implementation. However, the point update cannot assume that the last region was split in two by an edge being added, and needs to be able to trace both upwards and downwards, so there is a fair amount of work to be done to write this correctly. Timing data shows this to have a negligible effect on polygons up to at least 1500 vertices.

There are a few other issues that are still not in final working order. If both points of an edge must be added in, they are always added top point first, not in random order as they ought to be. This does not change asymptotic running time, but might worsen actual running time slightly. Also, vertices that should be added on top of an already added edge are added in a region next to the edge. Currently the vertices are added as if they were sitting next to the edge, and not on top, i.e., in the final trapezoidation, it will be possible to move up and down through the neighbors of the regions trapezoids on both sides of the vertex. In figure 9, it should not be possible to move up and down to the left of the point since, technically, there is nothing between the point and the line. This will not change the final trapezoidation or triangulation, but it might affect the depth of the regions, although I have still not seen this happen in any triangulations. Lastly, some of the manipulations of the random edge ordering in step 1 is done with lists. To get each different polygon to have one of its edges added among the first k edges, some runs through these lists are performed, and constant lookup time must be assumed to ensure this is linear. These lists should probably be changed into arrays to accommodate this. However, timing data shows that the randomization is entirely negligible for practical values of n, so this has still not been implemented.

8 Practical Results from the Mathematica Implementation

This routine was not written in order to perform as fast as possible, it was implemented in order to perform tests and experiments with it. Some of these experiments have resulted in an optimized and generalized algorithm, as discussed in earlier sections. This section shows the actual performance of the implementation, and how it compares to similar already existing routines in Mathematica.

Wolfram Research has created a $O(n \log n)$ polygon triangulation routine (see links). Their implementation uses Mathematica list structures instead of trees, so actual asymptotic performance would be $O(n^2)$, although with very small constants, so for most practical values of n the performance is close to $O(n \log n)$. Stan Wagon has implemented the $O(n^3)$ recursive ear clipping routine. Figure 12 shows how these routines compare for random polygons[7]. My implementation uses significantly more memory than either of the other ones, and at about $n = 1500$ the computer the tests were performed on starts using swap files for the memory, and time comparisons would no longer be valid. For polygons of this size the fastest routine is clearly the $O(n \log n)$ one, but at a size of about 3000

[6] In the actual code this procedure is called *UpdateCR*.

[7] Random here refers to polygons turned at random angles. All three routines perform differently for the same polygon, depending on the angle it is input with.

vertices it starts increasing faster than my routine. Comparison to Mathematica's routine at a stage before intersections were introduced was much more favorable to my routine, which tells me that even though it is left as an option to the user to turn such checking off, the option is still not working perfectly.

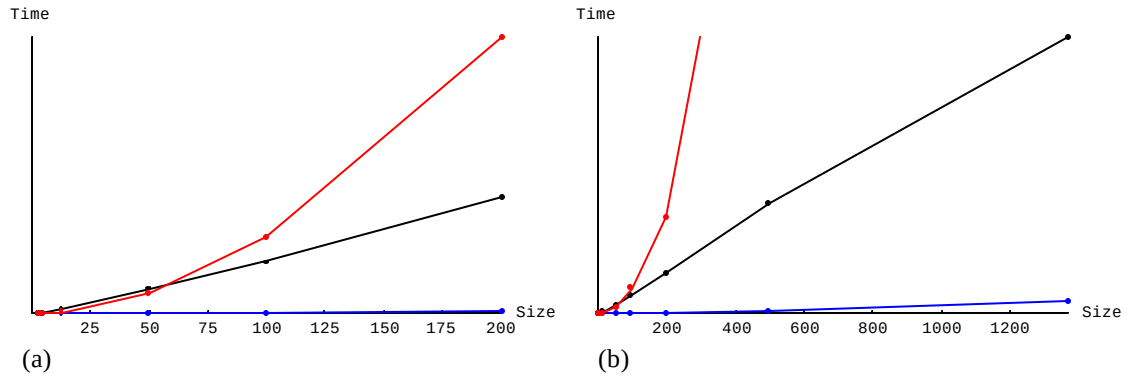


Figure 12: Comparison of different routines. Red is $O(n^3)$, Black is my implementation, and blue Mathematica's. Note the differing x-axis.

Figure 13 is a breakdown of the implementation. Magenta shows the time needed for creating a random order of the edges, while blue indicates the time taken for computing the trapezoidal decomposition of the polygon. The green line is a measure of how long it took to remove all the trapezoids on the outside of the polygon from consideration. The final steps of the algorithm, that of dividing the trapezoids into monotone mountain and triangulating them, both of which are done at the same time, is shown in red, and the total time is depicted in black. Also notice that the running time seems to be very close to linear in n and m, as should be expected.

The Mathematica implementation of polygon triangulation also contains a procedure to decompose non-simple polygons into a set of simple polygons. Figure 14 shows a comparison of this routine with my implementation. The polygons are generated as a list of random points in the unit square for this test. Note that this comparison is unfair to my routine, as Wolfram Research's routine only returns a set of non-intersecting polygons, while my routine triangulates them too. The horizontal axis is the number of edges plus the number of intersects, or m.

I was unable to get hold of another routine for triangulation of polygon with holes. Figure 15 is a test on a set of random polygons with a total of 60 vertices, where the polygons are allowed to intersect. The horizontal axis corresponds to the number of polygons. The triangulation time is relatively unchanged by the number of polygons. Each polygon is generated randomly as those for figure 14.

Human generated self-intersecting polygons can be quite different than random ones. For random polygons it almost never happens that two edges are overlapping, or that more than two edges intersect at the same point. This can happen quite often for human generated polygons. If polygons coordinates are restricted to integers only, the chance of degeneracies also increases. Testing shows that these polygons perform very similar to entirely random polygons. It is not possible to get a representative sample of non-random polygons, so a graphical comparison has not been done.

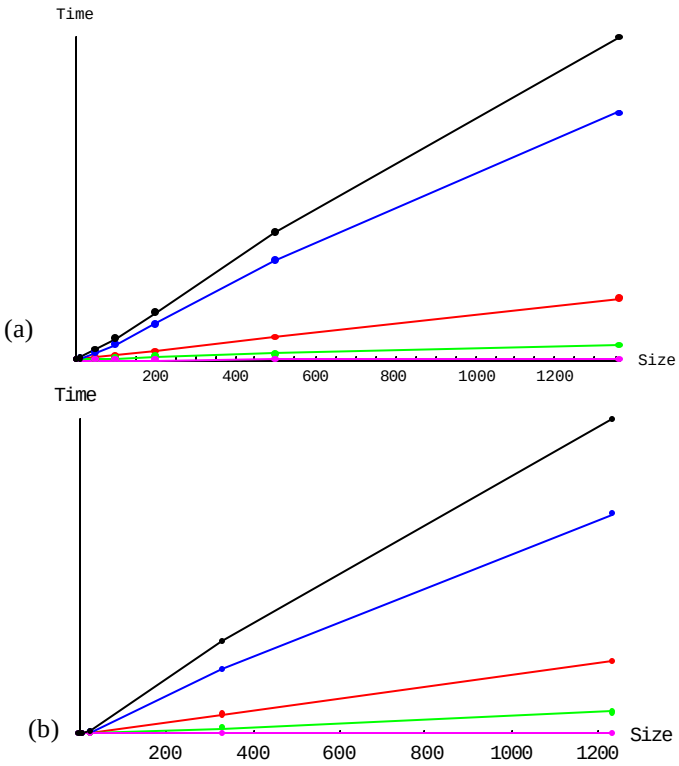


Figure 13: Breakdown of the routine.
Simple polygons (a) as a function of *n*,
and self-intersecting polygons (b) as a function of *m.*

Magenta: Randomization
Blue: Trapezoidation
Green: Removal of outside trapezoids
Red: Monotone mountain triangulation
Black: Total time

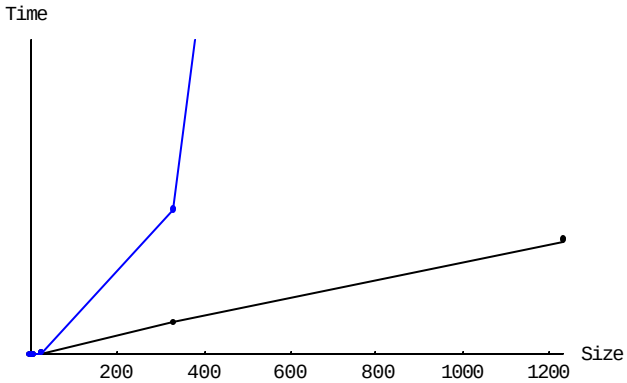


Figure 14: Comparison of decomposition of
self-intersecting polygons as a function of *m.*
Blue: Mathematica's polygon decomposition routine
Black: My polygon triangulation routine

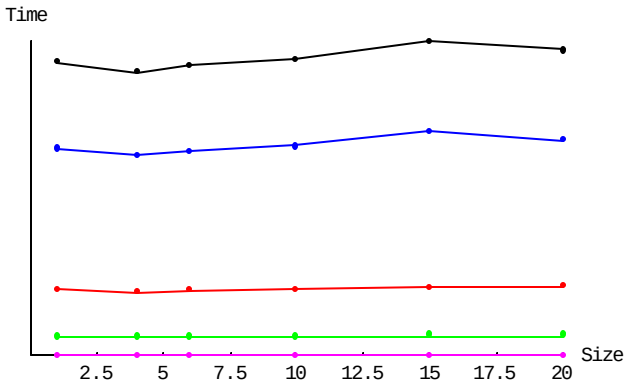


Figure 15: The effect of multiple polygons.
60 random vertices are divided into several polygons and triangulated. The horizontal axis is the number of polygons.

9 Notes on the implementation

This implementation is a tool for theoreticians, not a practical routine for fast triangulation. If you want to triangulate simple polygons in Mathematica, you should use Wolfram Research's own triangulation routine, although it performs an asymptotically slower line sweep routine in $O(n \log n)$, it works faster for most practical values of n. If you want to triangulate polygons industrially, you should use the code written by Narkhede and Manocha. They wrote a C++ implementation of the Seidel's algorithm, and although not quite as general as the one described here, it will be your best choice if you can use it.

The current version of the code is not entirely finished. It suffers from having only one developer and only one user. It also suffers significantly from limited time available for coding. Currently there is only limited commenting within the code, and Appendix C is all the documentation available. The size of the code is about 2000 lines, and was written in Mathematica 3.0. Mathematica 4.0 does run the code, it appears that it gives the correct output, although this is after a number of error messages have been printed, but extensive testing in this environment has not been done. If you want to use the code, you are recommended to use Mathematica 3.0.

There are many ways in which the code can still be improved. Some extra wrappers could be written, for instance to return all the triangles with the same orientation, or the make it possible to extract polygon by polygon (due to intersections these polygons might be different than the ones in the input) from *TriangulatedPolygon*. The triangulation can take polygons and single lines as input, but could be generalized to take strings of connected lines that do not necessarily form a polygon. It might also be of value to strip the triangulation of triangles of 0 area. Even if these are removed, the rest of the triangles will still describe the same polygons. Sometimes overlapping edges create trapezoids of 0 width between them, and these will in some cases end up as part of the final triangulation, depending on the position numbers of the vertices.

One particularly important aspect of the code that should be written would be some basic error checking on the input to the various routines. Right now they will try to work on whatever they are given, and if the input is incorrect or in the wrong order they will often crash with no hints as to what went wrong. Even though I know all the routines fairly well, I have spent quite a few hours trying to track down errors that were only caused by me giving incorrect input. Simple programming could check the input for validity, and tell the user how to do it correctly if the input is incorrect.

There are many ways in which this routine may be optimized. As the generalized algorithms were developed parallel to the implementation, the implementation was not always best suited for the final result, and has been changed many times. While programming, it often became evident how things could be done practically faster or more efficient, but not all of these ideas have been implemented. Some of the intersect parts of the code have not been gone through in detail after having been debugged, and I strongly suspect that some of them are still checking for possibilities that will never occur. Finally, the code could be updated to Mathematica 4.0 and optimized for this new environment.

10 Conclusion

The aim of this project was to study Seidel's algorithm in further detail, to implement it and try to generalize it and to compare the results to other algorithms. All of these results have been achieved. A working implementation, which to my knowledge is more general than any other implementation with a similar asymptotic time bound, has been written. Several extensions and optimizations to the algorithm have been developed and implemented. The expected running time of the final algorithm that has been developed is $O(m \log^* m + k \log k)$, where m is the number of edges plus the number of intersections, and k is the number of connected pieces.

One question remains: While the $O(k \log k)$ part is the minimum for k connected pieces, it might be possible to get the running time of triangulating a self-intersecting polygon down to linear time in m. Algorithms exist that can triangulate simple polygons in linear time, but it is not quite clear if either Chazelle's or Amato et al.'s routines could be enhanced to do a triangulation of a self-intersecting polygon in time linear in m. Both rely on coarse approximations of sub-chains of the polygon, and these approximations may no longer be valid if the sub-chains can intersect.

11 Acknowledgements

I would like to thank Alex Burst for all the work he has done on the routine. This originally started as a class project, on which we both did an extensive amount of work. His main responsibility was monotone mountain triangulation and graphical routines, and although all routines have been rewritten multiple times since then his work is still represented in the final code. My advisor Stan Wagon has helped me with optimizing Mathematica programming, and provided me with several routines of his that I have found very useful, a random polygon generation package in particular. He has also given me invaluable feedback on this paper. Finally I would like to thank Martin Kraus and Raimund Seidel for valuable discussions about the triangulation algorithm.

12 References

N.M. Amato, M.T. Goodrich and E.A. Ramos. Linear-time triangulation of a simple polygon made easier via randomization. *ACM Comput. Geom.* June 2000

T. Asano, T. Asano and R.Y. Pinter. Polygon triangulation: Efficiency and minimality. *J. Algorithms*, 7:221-231, 1986

B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6(5):485-524, 1991

B. Chazelle and J. Incerpi. Triangulation and shape-complexity. *ACM Trans. Graph.*, 3(2):135-152, 1984

A. Fournier and D.Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. Graph.*, 3(2):153-174, 1984

M.R. Garey, D.S. Johnson, F.P. Preparata and R.E. Tarjan. Triangulating a simple polygon. *Inform Process. Lett.*, 7(4):175-179, 1978

L.J. Guibas, J. Hershberger, D. Leven, M. Sharir and R.E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209-233, 1987

J. O' Rourke. Computational Geometry in C. *Cambridge Univ. Pr.*, 2nd ed., chapter 2, 1998

R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1(1):51-64, 1991

13 Links

This paper and its implementation

Currently undecided.

Will be made available on the web when work on the implementation has finished.

C++ implementation of Seidel's algorithm by Atul Narkhede and Dinesh Manocha at University of North Carolina at Chapel Hill

http://www.cs.unc.edu/~dm/CODE/GEM/chapter.html

Polygon triangulation package for Mathematica by Martin Kraus at Wolfram Research

http://library.wolfram.com/packages/polygontriangulation/