

The Triangulation of a Simple Polygon

The problem.

The triangulation of a simple polygon is a very old problem in computer science. The problem is stated in a very simple way, but it has proven to be difficult to develop an algorithm for a computer to solve it.

Given a simple polygon in two-dimensional space, we are to construct segments between the vertices of the polygon, such that the resulting figure is a collection of triangles only. A simple polygon means a polygon that does not have any intersecting edges. The polygon can be either convex or concave and may also have holes in it. Throughout this paper, whenever we give examples of simple polygons we will refer only to simple polygons that do not have any holes in them, although all the examples and algorithms can be easily modified to work with simple polygons with holes.

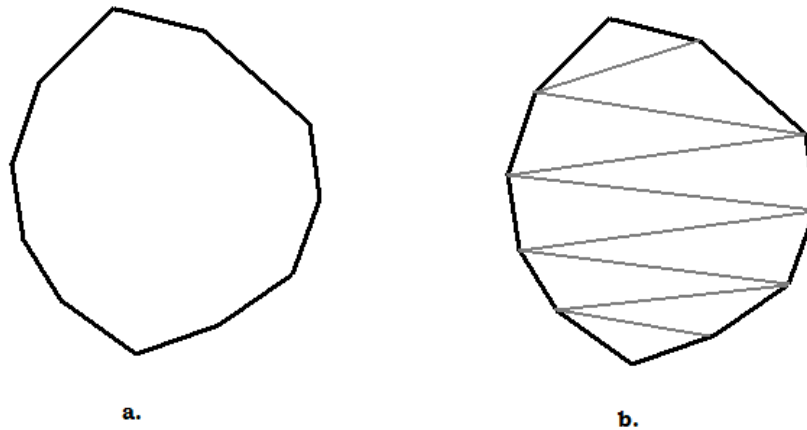


Figure 1. Example of a triangulation: a. original polygon; b. triangulated polygon.

The importance of an algorithm that accomplishes this task, besides the theoretical one, is that computers can, for most geometrical applications, work more easily with triangles than directly with polygons. For example, a computer can have dedicated hardware for computer graphics, which is more easily implemented for convex polygons (i.e., triangles) than for concave ones; it can easily compute shading of surfaces in three-dimensional space by computing shading in the two-dimensional space that does not depend on the orientation of the plane. Thus, triangles are the preferred way of decomposing the surface of a three-dimensional object, because they always are in a plane [3]. Having already computed the triangulation of a polygon, other problems, such as geometric decomposition, visibility, shortest path, separability, and subdivision preconditioning problems, are solved more easily [5]. Thus, one can immediately see the importance of this problem to computer scientists.

History of the problem.

The first algorithm for solving the problem in $O(n \log n)$, where n is the number of vertices of the polygon to be triangulated, was published in 1978 [4]. After the publication of this algorithm, computer scientists focused on either creating $O(n)$ algorithms for special kind of polygons, or creating $O(n \log k)$ algorithms for polygons that have special properties that are dependent on k . However, both types of algorithms have the disadvantage that they solve the problem only for a specific type of polygons, and, in the second case, the worst case running time is still $O(n \log n)$ [3]. Scientists were also faced with the question of whether there was any algorithm that could run in $O(n)$ time.

The answer came from Fournier and Montuno [3] who, in 1984, showed that the triangulation problem can be solved in $O(n)$ time by using a process they call *trapezoidization*. Although the triangulation itself worked in $O(n)$ time, the algorithm depended on the trapezoidization method, which was a $O(n \log n)$ algorithm, thus making the whole algorithm run in $O(n \log n)$ time. They also proved that any triangulation algorithm of (not necessarily simple) polygons required $O(n \log n)$ time, but there was no known lower bound on the triangulation of simple polygons. After the publication of Fournier and Montuno's paper, several other authors published different algorithms that produced the same results. The advantage of their algorithms was that they could be adapted to solve other problems in linear time. One such algorithm was given by Tarjan and van Wyk [6] in 1986.

After the late 1980's, the focus shifted from the triangulation problem to other, more general problems such as the triangulation of three dimensional surfaces and weighted-graphs. Several new algorithms for the original triangulation problem were developed during the 1990's, some that have tried to simplify the complexity of the data structures involved in solving the problem in $O(n \log n)$ time, at the expense of some speed, and some that have tried to improve the running time by using randomized algorithms. For example in 1990, Kirkpatrick, Klawe and Tarjan [5] solved the problem in "almost" linear time by using simple data structures, not the Jordan sorting and the finger search trees used by the previous algorithm [5]. Their algorithm runs in $O(n \log \log n)$ time. In 1991, Chazelle [2] announced a deterministic algorithm that runs in $O(n \log^* n)$. Although a great result, it was very complicated to implement. In 2000, Amato, Goodrich, and Ramos [1] finally discovered a randomized algorithm that runs in $O(n \log^* n)$ time, but which did not use complicated data structures, and thus was considered better and easier to implement than Chazelle's.

A simple outline.

In this paper we will focus more on the Fournier and Montuno [3] article, because it was the important milestone in the triangulation problem, the one that started the high interest of lowering the time complexity. The other algorithms were important too at the time of their publication, but we are choosing this particular algorithm because it is easier to understand and it provides some general concepts with which all the other algorithms work (i.e., trapezoidization). Afterwards, we will discuss some of the later algorithms that address the problems of the Fournier and Montuno algorithm. We will not refer as much to these algorithms, because they are too complex to explain for the purpose of this paper,

but we will mention some of the improvements that they brought to the solution of the problem.

Fournier and Montuno.

When Fournier and Montuno published their algorithm, they claimed that it was not the first algorithm to solve the problem of triangulating a simple polygon in $O(n \log n)$ time, but it was an algorithm that made use of simpler data structures and a simpler implementation than the ones before it to solve the problem.

The algorithm makes use of a process called trapezoidization, which tries to solve the following problem:

Find a minimal set of disjoint trapezoids that cover the polygon.

Without loss of generality, we will consider only trapezoids that have their parallel edges parallel to the other trapezoids' parallel edges and also parallel to the x axis (i.e., horizontal). These trapezoids will have edges that are part of the polygon as their non-parallel edges, and also each parallel edge will have at least one vertex from the original polygon.

Fournier and Montuno categorize all of the vertices of a polygon in three categories, depending on their adjacent edges' position relative to the horizontal line going through the vertex:

- *Type 1*: These are vertices with the adjacent edges on both sides of the horizontal line, and they mark the end of a trapezoid and the beginning of another one.
- *Type 2*: These are vertices with both edges below the horizontal line. If the inside of the polygon is above the horizontal line, then this type of vertex will mark the end of one trapezoid and the beginning of two new ones, otherwise, it will mark just the beginning of one trapezoid (the trapezoid will be just a degenerate case because it will look like a triangle).
- *Type 3*: These are vertices with both edges above the horizontal line. If the inside of the polygon is below the horizontal line, then this type of vertex will mark the end of two trapezoids and the beginning of another, otherwise, it will mark just the end of one trapezoid (the trapezoid will be just a degenerate case because it will look like a triangle).

For all these categories, and the ones that follow, we are considering a top-down scan of the polygon.



Figure 2. Types of vertices: a. Type 1; b. Type 2; c. Type 3.

Now we are ready to give the algorithm for the trapezoidization of a simple polygon. The problem is formally stated as follows [3]:

Input: A polygon $P(v_0:v_{n-1})$ with vertices given in clockwise order.

Output: A trapezoidized polygon $TP(v_0:v_{n-1})$. In order to define a trapezoid, each vertex v_i will have pointers to the second vertex v_j that defines the trapezoid, and to the left and right edges of the trapezoid e_{left} and e_{right} . Type 2 vertices might have two trapezoids associated with them, in which case the leftmost trapezoid is defined first, while Type 3 vertices might have none.

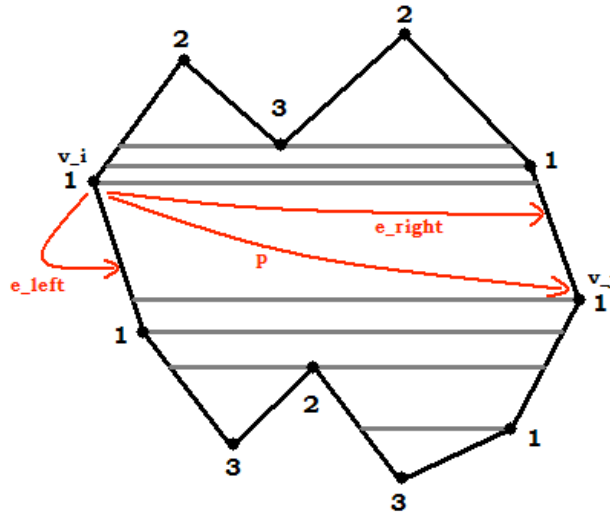


Figure 3. A trapezoidized polygon and the data structure associated with one vertex. The numbers show the type of vertex; v_i is the topmost vertex in the trapezoid; v_j is the vertex being processed; e_{right} and e_{left} are pointers to the right and left edges of the trapezoid respectively; p is a pointer to the other polygon vertex of the trapezoid.

The algorithm uses a 2-3 tree, a special case of a B-Tree, holding the *active* trapezoids as leaves (a trapezoid is active if intersected by a horizontal line between the last processed vertex and the next one). The right and left edges of the trapezoid are used as keys in searching. The algorithm uses the fact that while a trapezoid is active, a vertex is inside the trapezoid if and only if it is inside its left and right edge [3].

Trapezoidize(P)

Sort all vertices in decreasing order of y coordinates and increasing order of x coordinates for vertices of equal y coordinates.

For each vertex v_i in sorted order do

Determine type of v_i :

Type 1:

Find edge in 2-3 tree

Add v_i to complete trapezoid structure

Remove trapezoid from 2-3 tree

Replace edge to which v_i is adjacent

by other adjacent edge of v_i

Insert new trapezoid structure with v_i and the

right and left edges

Type 2:

Search for location in 2-3 tree

```

    If  $v_i$  is within an active trapezoid then
        Add  $v_i$  to complete trapezoid
        Remove trapezoid from 2-3 tree
        Insert two new trapezoid structures with
         $v_i$  and its edges in addition to the former
        trapezoid edges
    else
        Insert new trapezoid with  $v_i$  and its
        two edges
Type 3:
    Find adjacent edges in 2-3 tree
    If edges belong to same trapezoid then
        Complete trapezoid by adding  $v_i$ 
        Remove trapezoid from 2-3 tree
    Else
        Complete right and left trapezoids by
        adding  $v_i$  to their structures
        Remove both trapezoids from 2-3 tree
        Insert new trapezoid with  $v_i$  and the
        left trapezoid left edge and the right
        trapezoid right edge

```

The pre-sorting in the above algorithm can be done with any well known sorting algorithm, so it will take $O(n \log n)$ time. The main loop of the algorithm takes n steps. Inside the main loop, every operation takes constant time except the searches through the 2-3 tree, which might take as long as $O(\log n)$ in the worst case. Thus, the total running time of the algorithm is $O(n \log n)$. During the first iteration of the main loop, the first (topmost) vertex v_i found will be of type 2, and it will not be within an active trapezoid, so a new incomplete trapezoid structure will be inserted in the 2-3 tree, containing the vertex v_i and its two edges. During the last iteration of the main loop, the last (bottommost) vertex v_i found will be of type 3, and both its edges will belong to the same trapezoid, so v_i will be added to the trapezoid structure found in the 2-3 tree and the trapezoid structure will be removed from the tree. For all other iterations, vertex v_i will be either of type 1, in which case v_i will be added to complete the trapezoid structure found in the 2-3 tree, the structure will be removed from the tree, and a new incomplete trapezoid structure, containing v_i and its other edge, will be added in the tree; either of type 2, in which case v_i will be added to complete the trapezoid structure found in the 2-3 tree, the structure will be removed from the tree, and two new incomplete trapezoid structures, containing v_i and its two other edges, will be added in the tree; or type 3, in which case v_i will be added to both trees found in the 2-3 tree to complete the trapezoid structure, both trapezoids will be removed from tree, and a new incomplete trapezoid structure, containing v_i and the above left trapezoid's left edge and the above right trapezoid's right edge, will be added to the tree.

The correctness of the algorithm is obvious, so we will not prove it here. In implementing the algorithm there are some minor technicalities that need to be taken into consideration, things such as determining the correct type of a vertex when an adjacent edge is horizontal, but these details should not prove difficult to solve, considering that the vertices are sorted before the main loop. It should be noted that Fournier and Montuno do not say that this is a lower bound on the time it takes to trapezoidize a simple polygon.

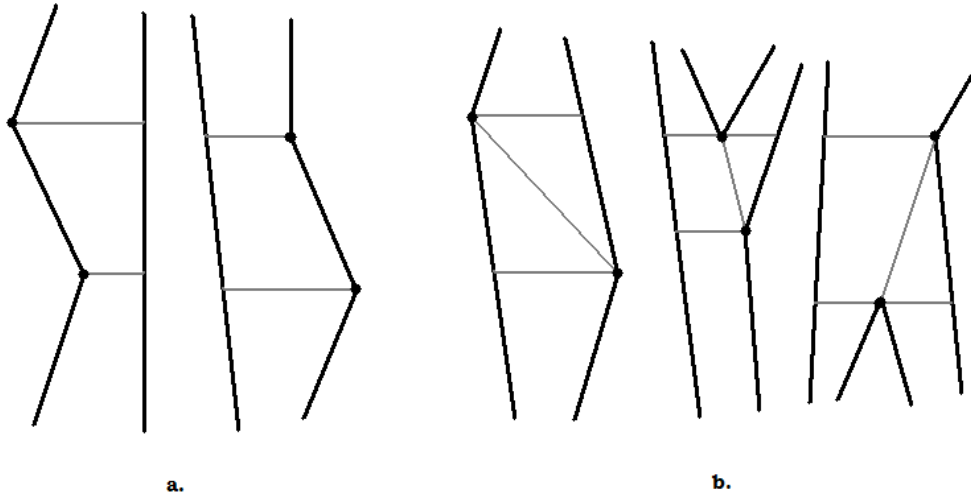


Figure 4. Classes of trapezoids: a. Class A; b. Class B.

Now we arrive at the problem of actually triangulating the polygon. We observe that the trapezoids can be classified into two classes [3]:

- *Class A*: These are trapezoids in which two polygon vertices share the same edge.
- *Class B*: These are trapezoids in which two polygon vertices do not share the same edge.

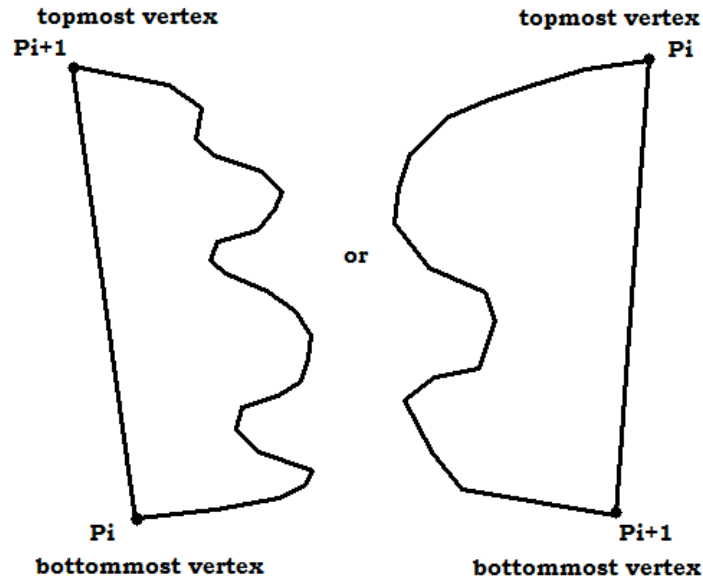


Figure 5. Unimonotone polygons.

It is easy to triangulate the Class B trapezoids by creating an edge between the two vertices of the polygon. After we have triangulated all Class B trapezoids, we are left with subpolygons that are *unimonotone* with respect to the y axis. Fournier and Montuno define a *unimonotone* polygon as a polygon $(P_0, P_1, \dots, P_{n-1})$ for which there is an i such that P_i and P_{i+1} are the vertices with minimum and maximum y coordinates (either order) and the other vertices are in non-decreasing or non-increasing order of y coordinates. They also give an algorithm that can triangulate a *unimonotone* polygon in linear time.

Now let us look at the algorithm proposed to triangulate the trapezoidized polygon. This problem is formally defined as follows [3]:

Input: Same as output of algorithm above.

Output: Same as algorithm above and with each vertex pointing to its list of adjacent triangles.

```

Triangulate(first, last)
  Current_vertex = first
  While not current_vertex.done do
    Current_vertex.done = true
    Bottom_vertex = diagonal(current_vertex)
    If bottom_vertex != nil then
      Save_next = next(current_vertex)
      Save_prev = prev(bottom_vertex)
      Next(current_vertex) = bottom_vertex
      Prev(bottom_vertex) = current_vertex
      Trapezoid(current_vertex) = nil
      Triangulate(bottom_vertex, current_vertex)
      Current_vertex.done = false
      Bottom_vertex.done = false
      Next(current_vertex) = save_next
      Prev(bottom_vertex) = save_prev
      Next(bottom_vertex) = current_vertex
      Prev(current_vertex) = bottom_vertex
      Triangulate(current_vertex, bottom_vertex)
      Return
    Else
      Current_vertex = next(current_vertex)
  Triangulate_monotone(first, last)

```

The function `diagonal(v)` returns the bottom vertex in the trapezoid associated with vertex v , that does not share the same edge. In this case, it also removes the trapezoid structure from v . In case the two vertices share the same edge, or if there is no trapezoid associated with v then the function returns `nil`. The call to `triangulate_monotone(first, last)` is done after the main loop has been executed and, thus, after the polygon `(first, last)` has been transformed into a *unimonotone* polygon.

We can prove by contradiction that `triangulate_monotone(first, last)` is called after `(first, last)` is a *unimonotone* polygon. Suppose that `(first, last)` is not a unimonotone polygon and that there is at least one pair of vertices between *first* and *last* whose y coordinates are out of order. Then there has to be at least one vertex of type 2 or 3. This means that the trapezoid associated with this vertex will be of class B and that `diagonal(v)` will return a non-`nil` pointer for this vertex. This means that the algorithm will start executing the code for the `if` statement and thus return without ever reaching the `triangulate_monotone(first, last)` call. This is a contradiction.

Here is the pseudo-code for the `triangulate_monotone()`.

```

Triangulate_monotone(first, last)
  Determine start vertex (topmost if the monotone chain is on the
  right, bottommost if it is on the left)
  Determine number_of_vertices
  Current = next(start)
  While number_of_vertices >= 3 do

```

```

If angle(prev(current), current, next(current)) is convex
  For each of prev(current), current, next(current) do
    Insert other 2 vertices to form a triangle
  Save = prev(current)
  Remove current from unimonotone polygon
  If current = first then
    Current = next(first)
  Else
    Current = save
  Decrement number_of_vertices
Else
  Current = next(current)

```

The initialization steps take at most $O(n)$ time, since the vertices are already sorted. Because *unimonotone* polygons have at least one angle that is convex besides the topmost and bottommost ones the condition inside the while loop will be satisfied at least once. There cannot be other vertices between the current vertex and its adjacent vertices because this would violate the *unimonotone* property of the polygon. After the triangle formed from the current vertex and its adjacent vertices is created in the for loop, the current vertex is not considered as being part of the *unimonotone* polygon any more, and thus a new *unimonotone* polygon is created. By induction, this polygon will again have a convex angle besides the topmost and the bottommost angles and the condition in the while loop will be satisfied again. This will repeat until the number of vertices remaining in the *unimonotone* polygon will be 3, in which case the triangulation is already made and the function returns. The algorithm goes back one step only when a vertex is removed, so there are at most $O(n)$ backwards steps and at most $O(n)$ forward steps in the while loop.

Therefore:

Unimonotone polygons can be triangulated in $O(n)$ steps.

Since the triangulate algorithm breaks polygons into unimonotone polygons in $O(n)$ time, the following is true:

Trapezoidized polygons can be triangulated in $O(n)$ steps.

And thus:

Simple polygons can be triangulated in $O(n \log n)$ time.

Fournier and Montuno also talked in their paper about the necessary modifications to these algorithms in order to make them work for simple polygons with holes (which themselves can be simple polygons with holes). As the authors pointed out, their approach to triangulation did not improve on the upper bound of the algorithm, but it did provide programmers with an algorithm that did not need complicated data structures, that used a basic algorithm (the trapezoidization, also known as the scan conversion algorithm), and which used only mathematical computations and comparisons to check for vertices in the 2-3 tree and checking for angle convexity. In this case angle convexity is checked by using the cross product of the matrices associated with each of the adjacent edges. Moreover, their paper proved that the triangulation problem is linear-time reducible to trapezoidization.

Tarjan and van Wyk.

It is easy to understand why Fournier and Montuno's result was so important in computer science. All scientists had to do was to find a lower bound on the trapezoidization problem, in order to find a lower bound on the triangulation problem.

The first breakthrough came from Tarjan and van Wyk in 1986 [6]. They discovered a way in which one could trapezoidize (they call the process computing internal horizontal edge-vertex visibility) a simple polygon in $O(n)$ time, which led to a $O(n)$ triangulation algorithm. Although the result was important, the algorithm was very difficult to implement.

Tarjan and van Wyk used a modified version of Jordan sorting, which, at the time, had a known solution that ran in linear time. The Jordan sorting problem is stated this way: given k points at which the edges of a polygon intersect a horizontal line, in the order in which they are encountered in a traversal of the boundary of the polygon, sort them into the order in which they appear along the line. After this process is over, one has the edge-edge visibility information of a particular polygon along the given horizontal line. Tarjan and van Wyk showed that Jordan sorting (a linear time process) was linear-time reducible to computing all edge-vertex and edge-edge visibility information (i.e., trapezoidization). That implied that the triangulation problem was at least as hard as the Jordan sorting problem, thus having a linear time solution. The authors' approach was to use Jordan sorting with a divide-and-conquer method to solve the triangulation problem.

One technicality of their approach was that, besides implementing a variation of Jordan sorting with divide-and-conquer, one also had to implement recursive finger search trees, a data structure that is particularly difficult to implement, but which was key to the linear time solution. Scientists were not satisfied with the result and further research was conducted.

Improved solutions.

In 1991, Chazelle developed an algorithm that optimally and deterministically solved the triangulation problem in $O(n \log^* n)$ time. The result followed similar results from Clarkson *et al.* (1991) and Seidel (1991) who both gave algorithms with the same time complexity, but which were randomized, non-optimal solutions and were very difficult to implement. Although these results preceded Chazelle's, throughout the history of the problem, Chazelle's contribution is considered the most important step, being the first *optimal, deterministic* algorithm to solve the problem in *almost* linear time. Although the algorithm is technically not linear, for all practical purposes, it can be considered linear because the function $\log^*(n)$ is a very slow growing function.

The next breakthrough was achieved by Amato, Goodrich and Ramos [1] who published in 2000 a randomized algorithm that solved the problem in $O(n)$ time and did not use complicated data structures. Although their algorithm was a relative improvement over Chazelle's solution, using simpler data structures, it was not a big improvement over Clarkson's and Seidel's non-optimal solutions, as it was significantly more complicated to implement. Moreover, the time improvement over Chazelle's solution was insignificant, because, as we have mentioned before, for all practical purposes, $O(n)$ is almost the same as $O(n \log^* n)$.

Conclusion.

As Amato, Goodrich and Ramos said, one of the open questions in computer science remains whether there is any way of solving the triangulation problem optimally and deterministically with a method easier to implement than Chazelle's algorithm.

References:

1. N. M. Amato, M. T. Goodrich, and E. A. Ramos, "Linear-Time triangulation of a simple polygon made easier via randomization," *Proc. 16th Annu. ACM Sympos. Comput. Geom.*, pp. 201-212 (2000).
2. B. Chazelle, "Efficient Polygon Triangulation," preprint.
3. A. Fournier and D. Y. Montuno, "Triangulating simple polygons and equivalent problems," *ACM Trans. on Graphics* **3**(2), pp. 153-174 (1984).
4. M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, "Triangulating a simple polygon," *Inf. Proc. Lett.* **7**, 4, pp. 175-179 (1978).
5. D. G. Kirkpatrick, M. M. Klawe, and R. E. Tarjan, "Polygon triangulation in $O(n \log \log n)$ time with simple data-structures," *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pp. 34-43 (1990).
6. R. E. Tarjan and C. J. van Wyk, "A Linear-Time Algorithm for Triangulating Simple Polygons," *Proc. 18th Annu. ACM Sympos. Theory of Comput.*, pp. 380-388 (1986).