

**CS 478**

**Computational Geometry**

Final Report

*Three Delaunay Triangulation*

**Group Members**

Ahmet Faruk Ulutaş - 21803717

Mustafa Azyoksul - 21501426

13.05.2022

# 1. Introduction

In this project, we have implemented 2 different algorithms to solve Delaunay Triangulation (DT) problem. The 2 algorithms we study are Randomized Incremental Delaunay Triangulation and Divide and Conquer Delaunay Triangulation algorithms.

We have also programmed a GUI application that allows users to interact with these 2 algorithms. The user can choose to run the algorithm with any number of points, add/remove points at will, and visualize the triangulation process with a simulation.

# 2. Implementation Details

We have implemented 2 different algorithms to solve the DT problem. In this section, we discuss the implementation details and analysis of these implementations.

## 2.1. Random Incremental Algorithm

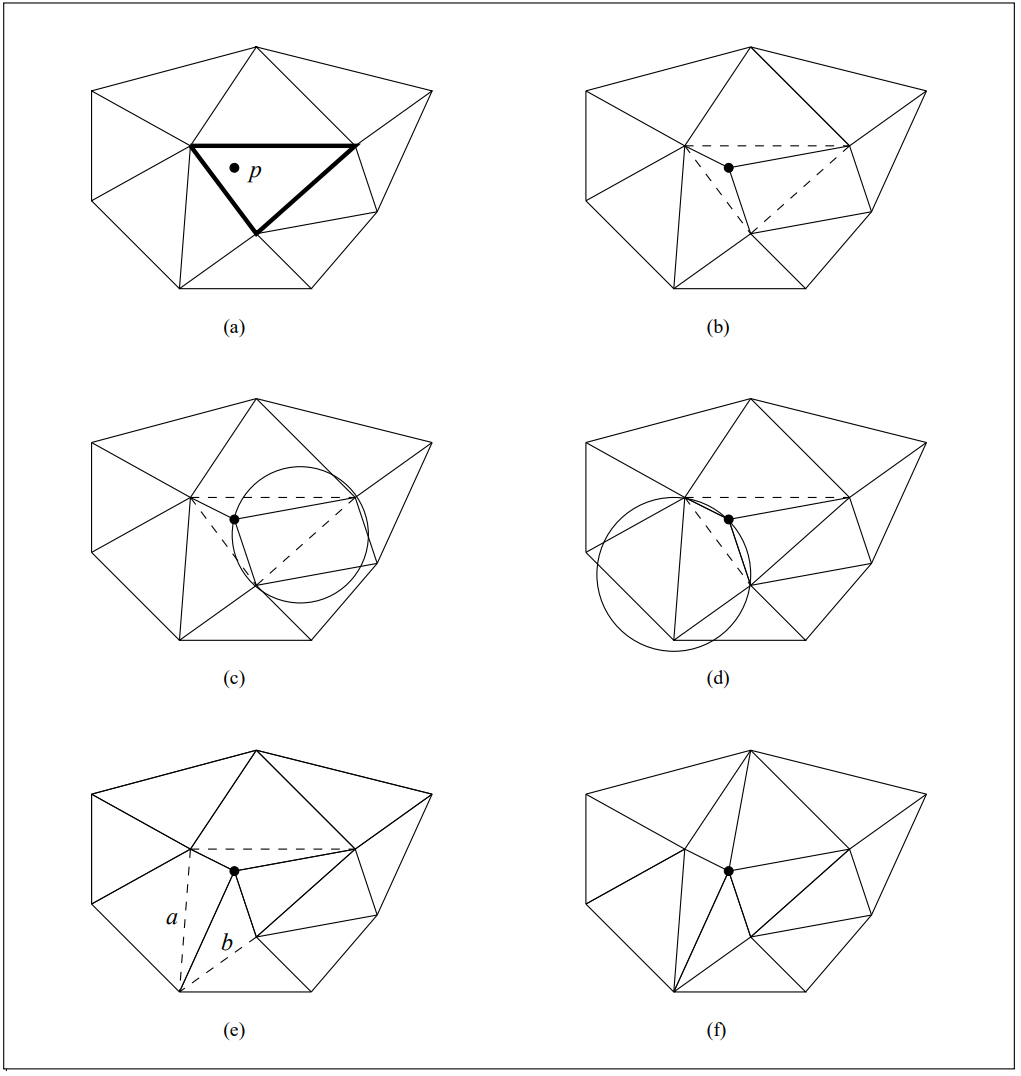
The aim of the random incremental algorithm is to add each vertex to the triangulation one by one and maintain valid DT properties at the end of each step. To achieve when an edge is added, this algorithm first creates a valid triangulation and then updates the current triangulation carefully so that the triangulation of all vertices added so far is a valid DT.

### Algorithm

First, a triangle that contains all points is found. The purpose of this triangle is to make the iterative process more streamlined. When an edge is added to the current structure, the first order of business is to locate which triangle it currently lies in. This way, finding a valid triangulation after an edge is added is straightforward. To achieve this, the algorithm needs to make sure that every vertex added is inside an already existing triangle. This requirement is met using the inclusive triangle added to the triangle list in the beginning.

It is important that this triangle is sufficiently large so that when it is ultimately removed, the remaining triangulation is valid. In other words, if a triangle T includes one of the inclusive triangle’s vertexes, the other two vertexes of T must be either on the convex hull of the currently processed point set or another vertex of the inclusive triangle.

After finding a suitable inclusive triangle and initializing the triangle set, we call a subprocedure that adds each point in the set into the triangulation one by one. After the additions are done, the inclusive triangle and all triangles that share a vertex with it are removed.



*Figure 1 [1]. Inserting a point into the triangulation. Dashed lines show the candidate edges.*

During the point addition subprocedure, the first step is to find which triangle of the current triangulation contains the given point. The most optimal implementation of this algorithm solves this first step asymptotically better than our implementation in this project which solves it in O(n) time. The second step is to remove this container triangle and form 3 new triangles that triangulate it. In the current state, we have a valid triangulation, however, it might not be a valid DT. There are 3 edges that might be violating the DT properties and those are the edges of the containing triangle. In the last step, we call another recursive subprocedure that legalizes these edges.

In the edge legalization subprocedure, the first step is to find the other triangle T that contains this edge. If T exists, the second step checks if the last vertex of T is inside the circumcircle of the previous triangle. If it is inside the circumcircle, the edge is illegal. To fix this, we swap the illegal edge. In this new triangulation, the other 2 edges of T could become illegal. Then as the last step, we call the edge legalization on the other two edges of T. Although it looks like in the worst case there can be O(n) recursive calls, in a 2D plane, because of the geometric properties of point locations, the number of recursive calls on this step is amortized at O(log n).

### Pseudocode

RandomIncrementalDT(P)

# Find a triangle that contains all the points

inclusive = findInclusiveTriangle

# Initialize the triangle list

triangles = [inclusive]

# Add points

for p in P

AddPoint(p, triangles)

# Remove all triangles that have a common vertex with the inclusive triangle

triangles.remove(inclusive)

return triangles;

AddPoint(point, triangles)

# Find which triangle contains the point

containing = FindContainingTriangle(point, triangles)

# Remove the containing triangle from the list of triangles

triangles.remove(containing)

# Form 3 new triangles

triangle1 = [containing[0], containing[1], point]

triangle2 = [containing[1], containing[2], point]

triangle3 = [containing[2], containing[0], point]

# Add 3 new triangles to the list of triangles

triangles.append(triangle1)

triangles.append(triangle2)

triangles.append(triangle3)

# Legalize the edges on a recursive call

legalize\_edge(point, triangle1, triangles)

legalize\_edge(point, triangle2, triangles)

legalize\_edge(point, triangle3, triangles)

LegalizeEdge(againts, container, triangles):

# Identify the edges

v1 = triangles.firstVertexNot(againts)

v2 = triangles.secondVertexNot(againts)

# Find the other triangle nearing the edge-to-be-legalized

for triangle in triangles

if v1 in triangle & v2 in triangle & against not in triangle

last = triangle.otherVertex(v1, v2)

# Check if this edge is illegal

if InCircle(v1, againts, v2, last)

# Edge swap

triangles.remove(triangle)

triangles.remove(container)

new\_t\_1 = [commonV1, againts, last\_vertex]

new\_t\_2 = [commonV2, againts, last\_vertex]

triangles.append(new\_t\_1)

triangles.append(new\_t\_2)

# Recursively legalize the new candidates

legalize\_edge(againts, new\_t\_1, triangles)

legalize\_edge(againts, new\_t\_2, triangles)

# Edge legalized

break loop

### Analysis

In theory, this is implemented in the most optimal way, the algorithm has a time complexity of O(n logn) [2]. This version involves a preprocessing step that will allow very fast lookup times to find this triangle a new point is in. It also uses an edge list data structure to represent the triangulation, not a triangle list structure.

In our implementation, the addition of every edge runs at O(n) time. In each point addition, there are 3 edge legalization steps. In the edge legalization step, finding in which triangle the point is in takes O(n) time. Then the recursive calls of edge legalization take O(logn) time with amortization. Overall, the algorithm’s **time complexity is O(n^2 logn)** = O(n) \* O(n) \* O(logn). This is not a very desirable time complexity. We do not recommend this implementation to be used in a practical scenario.

In the edge legalization procedure, constant space is used and this subprocedure is called O(n^2 logn) times. Therefore, the **space complexity is O(n^2 logn)**.

## 2.2. Divide And Conquer Algorithm

The aim of the divide and conquer algorithm is to apply a divide, triangulate, and merge strategy similar to the merge sort algorithm. Points are ordered by an axis. Subgroups of sizes 2 and 3 are trivial to triangulate. Then in the merge step, the combined DT of the left DT and right DT is calculated. The algorithm makes use of a very sophisticated data structure called the **quad edge** to significantly improve performance.

### Algorithm

This algorithm uses a very sophisticated directed edge data structure called **quad edge**. This structure holds the following information:

* Origin of the edge
* Destination of the edge
* Symmetric of the edge. This is a reference to another edge which is directed from the destination to the origin of this edge.
* Next neighbour. This is a reference to another edge which is the next neighbor of this edge on the edge ring in the counter-clockwise direction.
* Previous neighbor. This is a reference to another edge which is the next neighbor of this edge on the edge ring in the clockwise direction.
* Deleted flag

First, we preprocess the input. Preprocessing consists of validating the input size, removing duplicates, and sorting the edges by the x-axis first then by the y-axis on the ties. After preprocessing is done, we call the recursive div-and-conq subprocedure.

The div-and-conq is a conventional recursive call similar to the merge sort. First, we check for the base cases, which are when the size of the input set is 2 or 3. Then, we split the set of points into two and call the recursive div-and-conq on both of the subsets. Here, all points of the first subset have smaller x-axis values than all the points in the second subset. Then lastly, we call the merge process.

The merge process takes two linearly separable set of valid DTs and combines them into a single valid DT in linear time. To achieve the linear time complexity, it makes use of a few inputs from the previous recursive div-and-conq calls.

Our implementation is based on Guibas and J. Stolfi’s paper [4] and follows all the same conventions they have stated in the paper. We have borrowed the implementation of the merge procedure from an existing GitHub repo[3] which is also based on the same paper.

### Pseudocode

QuadEdge(from, to)

from = from # Origin

to = to # Destination

self.sym = empty # Same edge, backwards

nextCCW = this # Next edge on the edge ring

prevCW = this # Previous edge on the edge ring

deleted = false

div\_and\_conq\_delaunay(S)

# Validate the input size

if S.size < 2

return

S.removeDuplicates()

S.sort()

edges = []

div\_and\_conq\_triangulate(S, edges)

# Remove edges that are not part of the triangulation

return edges.removeDeleted()

div\_and\_conq\_triangulate(S, edges):

# Base case: 2 points

if S.size == 2

edge = create\_edge(S[0], S[1], edges)

return edge, edge.sym

# Base case: 3 points

elif len(S) == 3

# Create edge S[0]-S[1] and edge S[1]-S[2]

edge1 = create\_edge(S[0], S[1], edges)

edge2 = create\_edge(S[1], S[2], edges)

splice(edge1.sym, edge2)

# Create edge S[2]-S[0]

det = left\_test(S[2], [edge1.from\_, edge1.to])

# Right

if det < 0

connect(edge2, edge1, edges)

return edge1, edge2.sym

# Left

elif det > 0

edge3 = connect(edge2, edge1, edges)

return edge3.sym, edge3

# Points are linear

else

return edge1, edge2.sym

# Recurively triangulate the left and right halves

else

m = len(S) // 2

ldo, ldi = div\_and\_conq\_triangulate(S.sub(0,m), edges)

rdi, rdo = div\_and\_conq\_triangulate(S.sub(m, S.size()), edges)

ldo\_r, rdo\_r = merge(ldo, ldi, rdi, rdo, edges)

return ldo\_r, rdo\_r

merge(ldo, ldi, rdi, rdo, edges)

#Refer to the code for the implementation of the merge process

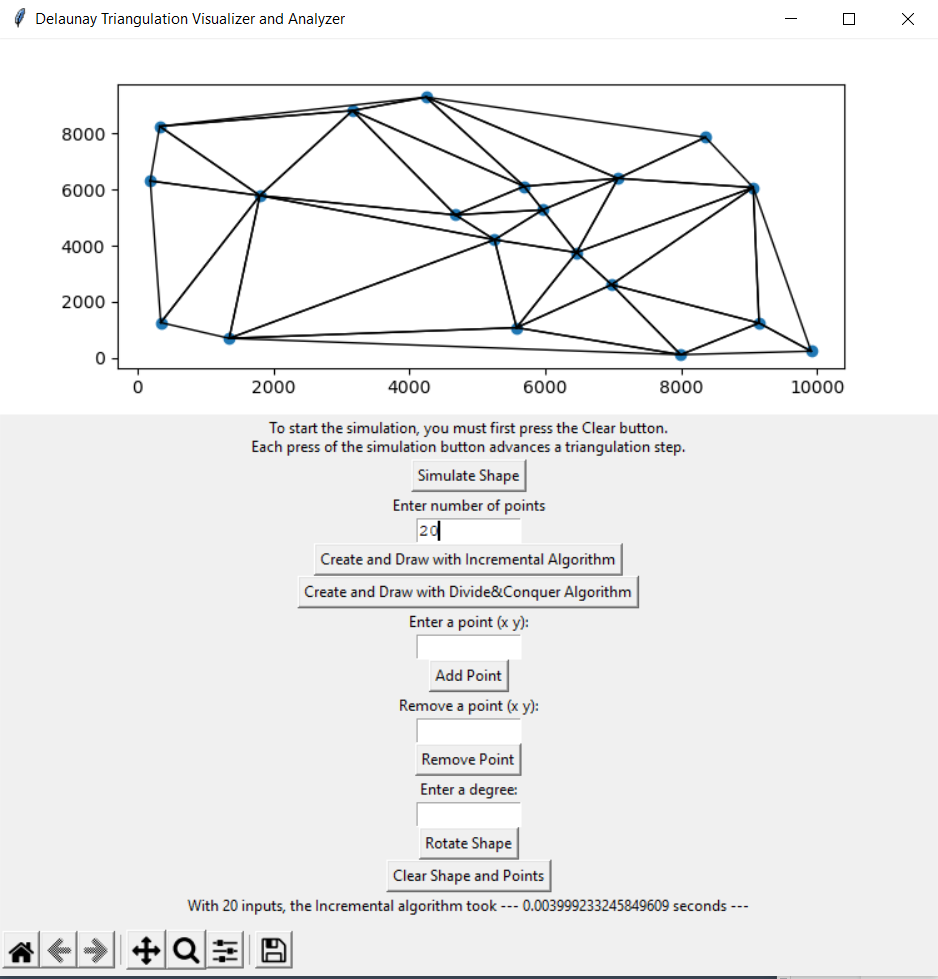
# Merge process performance is theoretically optimal. [3]

### Analysis

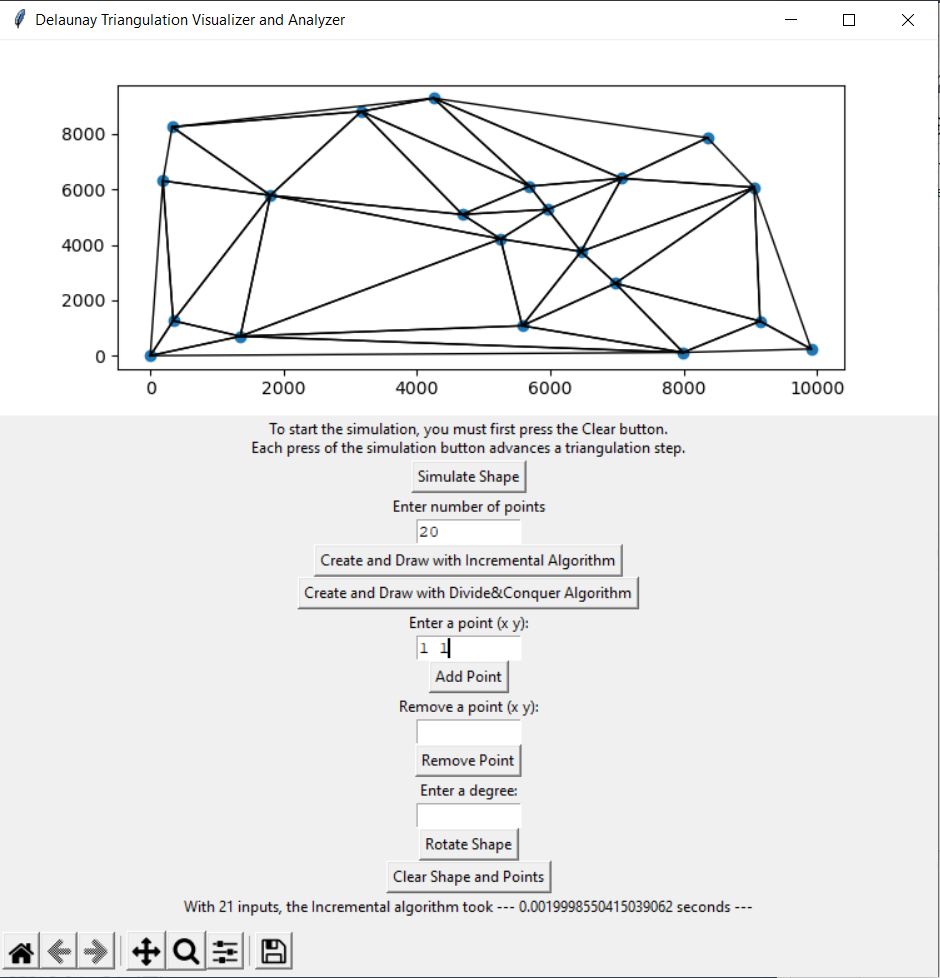
The recursive calls take O(logn) time and the merge process takes O(n) time. Overall the **time complexity is O(n logn)**. Similarly, the **space complexity is O(nlogn)**.

# 3. Program Features

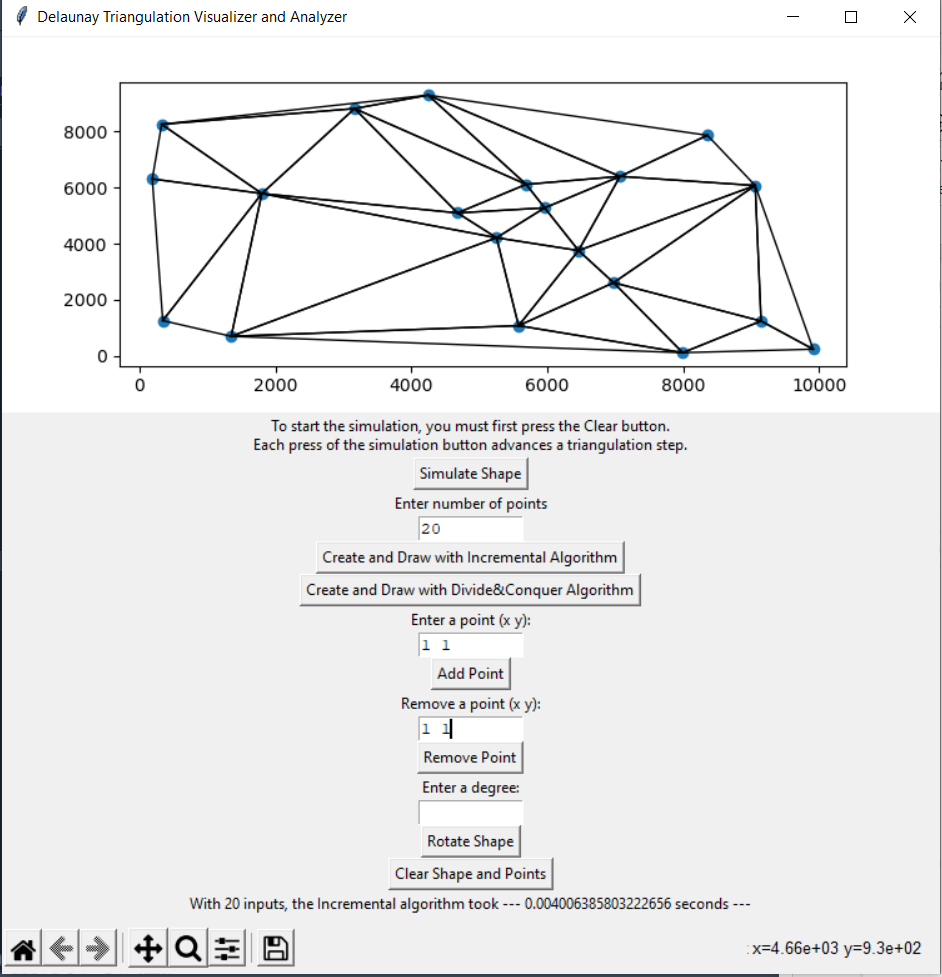
In our application, you can enter the number of points you specify by using any of the algorithms. You can add or delete points as desired. You can rotate the shape at any angle you want. When you take any action, the chart will be automatically updated accordingly. If you want to examine the formation steps of the Delaunay triangulation, you can examine all the steps from 0 one by one by pressing the simulate shape button. By using the bars at the bottom left of the page, you can use options such as reset, undo the operation, play the shape, zoom in, zoom out, and save. In addition to these, when you use any algorithm, there is information about the performance analysis of the algorithm and how many seconds it works.



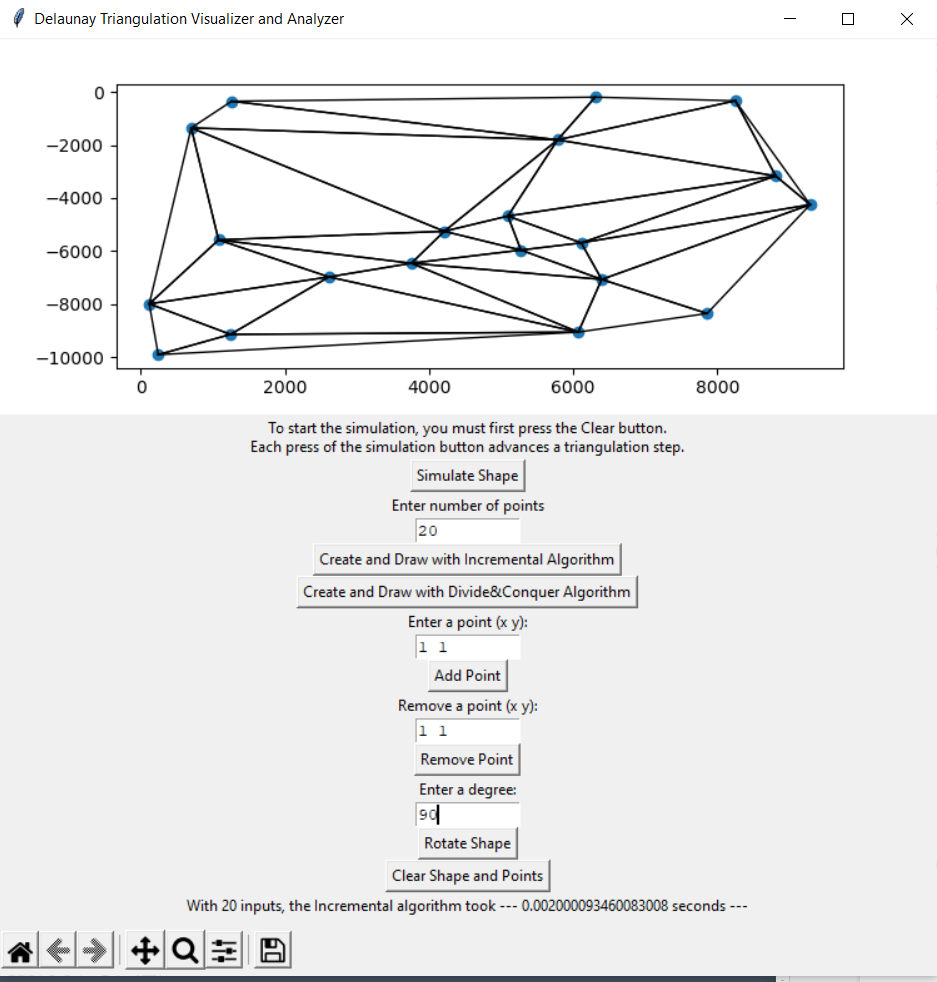
Generating a certain number of random points as an example



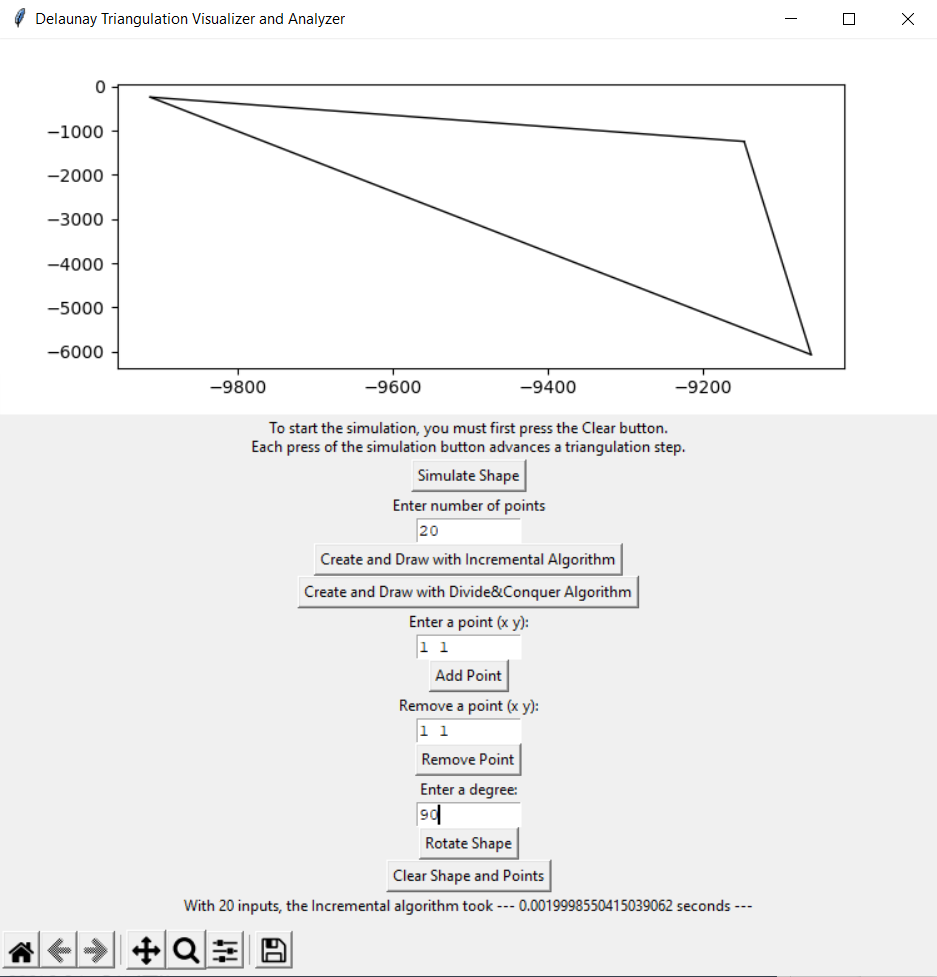
Adding a specific point with certain coordinates



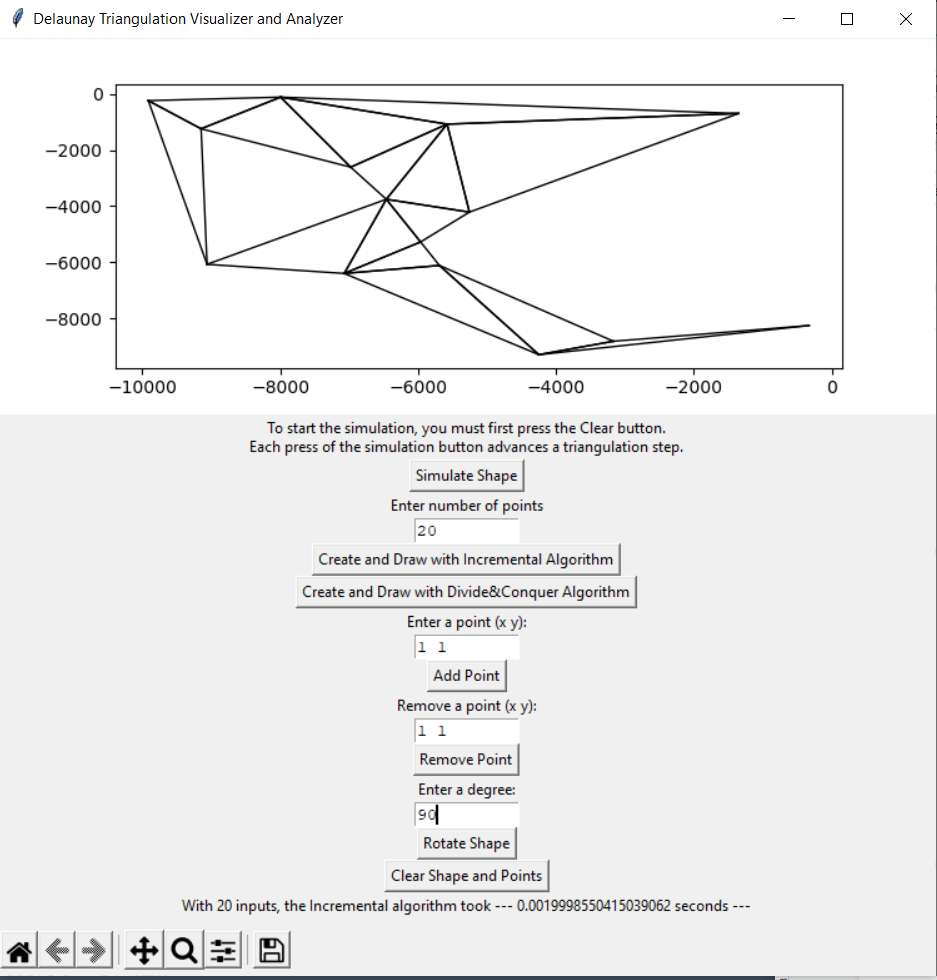
Deleting a specific point on triangulation



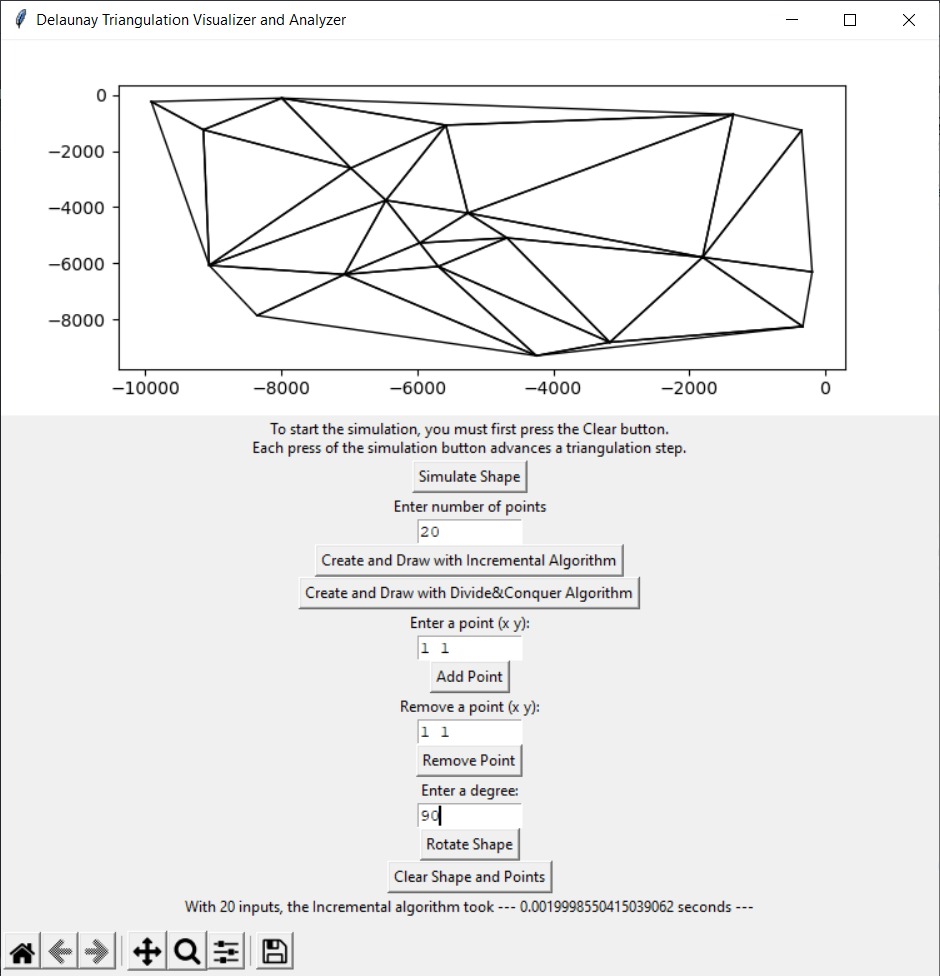
Rotate triangulation to desired degree



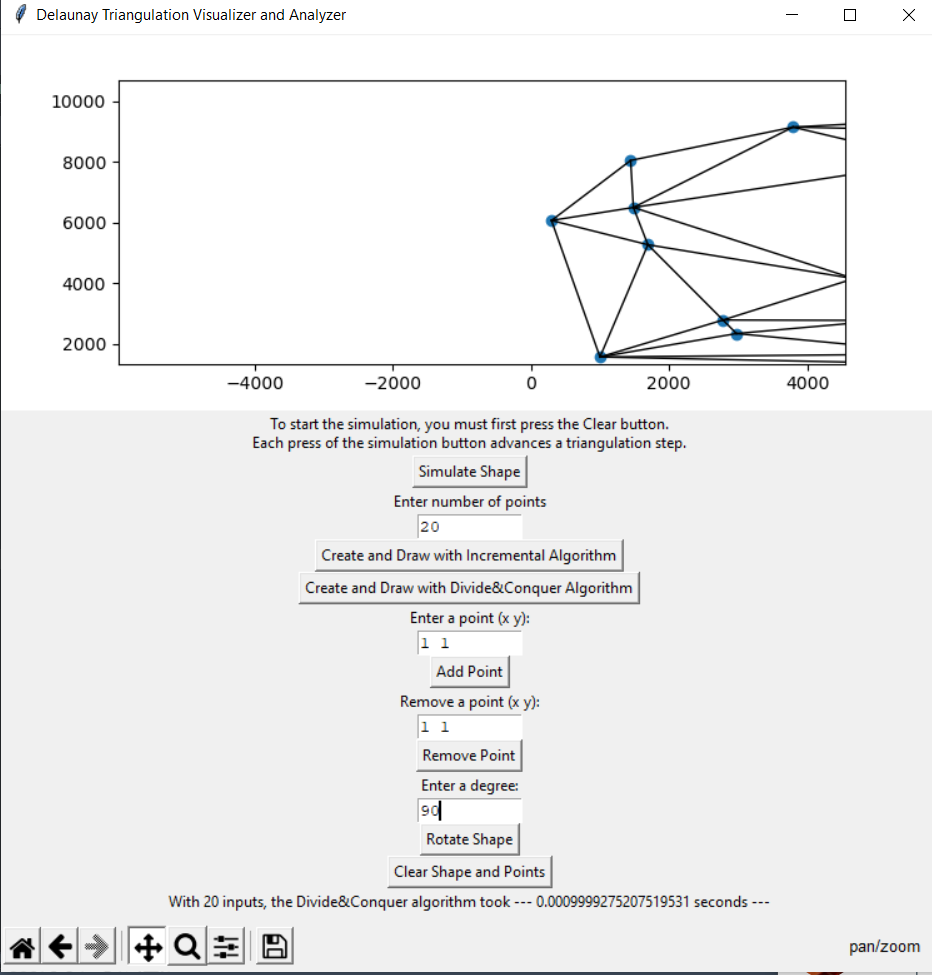
Monitoring the whole process step by step using the Simulate Shape button (step1)



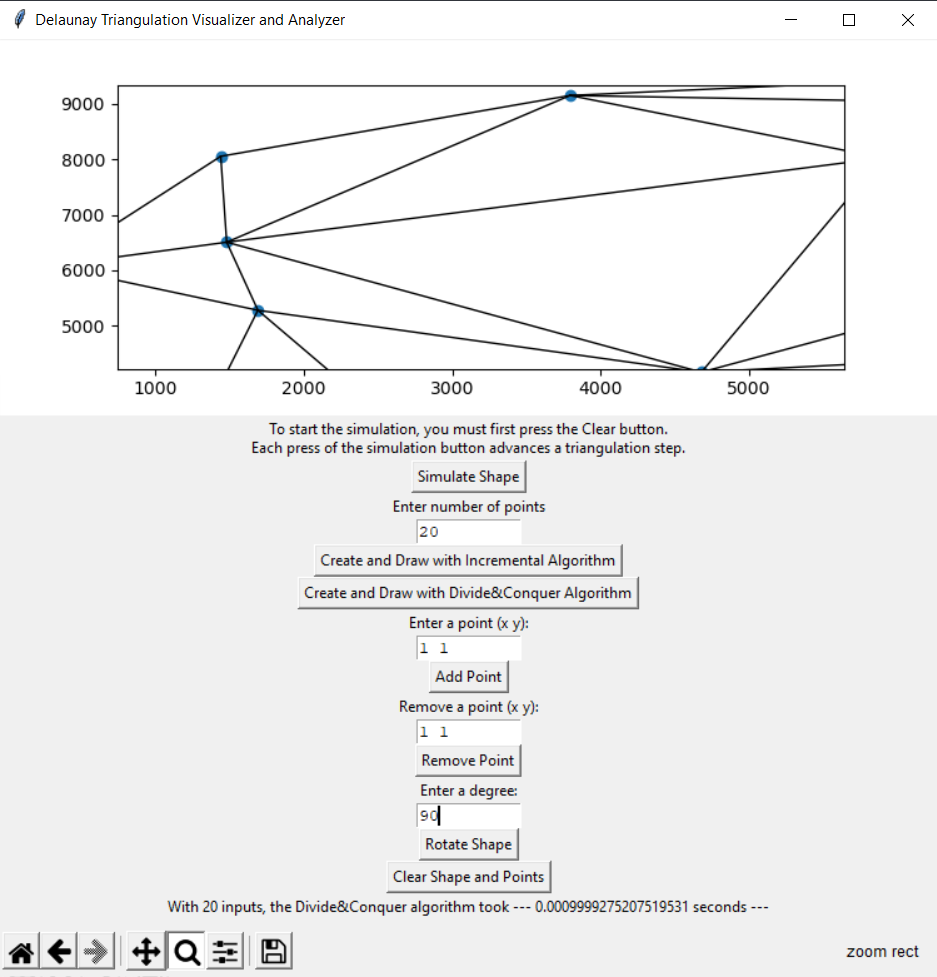
Monitoring the whole process step by step using the Simulate Shape button (...step10)



Monitoring the whole process step by step using the Simulate Shape button (...last step)



Moving the object to the desired location



Zoom to desired area on triangulation

# 4. Conclusion

We have attempted to implement both algorithms in the most optimal way. Divide and Conquer Delaunay Triangulation runs at the theoretical optimal, however, due to the suboptimal data structure used, Randomized Incremental Delaunay runs slower than the theoretical optimal.

# References

[1] Lischinski, Dani. “Incremental Delaunay Triangulation” Graphics Gems IV. Paul S. Heckbert (Ed.). Academic Press Prof., Inc., San Diego, CA, USA. 1994. Page 47-59. http://www.karlchenofhell.org/cppswp/lischinski.pdf doi: https://dl.acm.org/citation.cfm?id=180900

[2] “Triangulation - karlchenofhell.org.” [Online]. Available: http://karlchenofhell.org/cppswp/lischinski.pdf. [Accessed: 13-May-2022].

[3] Alexbaryzhikov, “Alexbaryzhikov/triangulation: Divide-and-conquer algorithm for Delaunay triangulation,” GitHub. [Online]. Available: https://github.com/alexbaryzhikov/triangulation. [Accessed: 02-May-2022].

[4] L. Guibas and J. Stolfi, “Primitives for the manipulation of general subdivisions and the computation of Voronoi,” *ACM Transactions on Graphics*, vol. 4, no. 2, pp. 74–123, 1985.

[5] S.T.E.V.E.N. FORTUNE, “Voronoi diagrams and Delaunay triangulations,” Lecture Notes Series on Computing, pp. 225–265, 1995.

[6] B. Delaunay, “Sur la sphère vide, Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestven- nykh Nauk,” pp. 793–800, 1934.

[7] Leach, Geo. "Improving worst-case optimal Delaunay triangulation algorithms." *4th Canadian Conference on Computational Geometry*. Vol. 2. 1992.

[8] Cignoni, Paolo & Montani, Claudio & Scopigno, Roberto. (1998). DeWall: A fast divide and conquer Delaunay triangulation algorithm in E d. Computer-Aided Design. 30. 333-341. 10.1016/S0010-4485(97)00082-1.

[9] S. Peterson, “Computing Constrained Delaunay Triangulations,” *COMPUTING CONSTRAINED DELAUNAY TRIANGULATIONS IN THE PLANE*. [Online]. Available: http://www.geom.uiuc.edu/~samuelp/del\_project.html#:~:text=The%20divide%20and%20conquer%20algorithm,by%20their%20y%2Dcoordinates). [Accessed: 22-Mar-2022].