Delaunay Triangulation and Voronoi Diagram

Karthik Iyer
UIN: 234009250

1 Abstract

A Delaunay Triangulation of a set of points is a triangulation such that no point is inside the circumcircle of any triangle in the triangulation. There can be many different triangulations for a set of points, but Delaunay triangulations maximize the minimum of all angles of the triangles. In other words, the triangulation tends to avoid thin triangles. The geometric properties of Delaunay Triangulations have many applications across various domains such as modeling terrain from a point cloud, generating meshes for finite element method, path planning, algorithmic art and more.

Delaunay Triangulation can also be used to derive a dual Voronoi diagram, which is a set of polygons with the circumcenters of the triangles as vertices. Voronoi diagrams also have a plethora of applications, such as calculating average precipitation at each weather station, modeling biological structures such as cells, algorithmic art, and architecture.

Through this project, I have explored methods to compute Delaunay Triangluation along with a practical implementation of the Bowyer-Watson Algorithm in C++. A lot of research has gone into finding efficient ways to generate triangulations conforming to the Delaunay criterion, but they can be broadly classified into the following categories: Incremental, Divide and Conquer, Sweepline, Gift wrapping, and Convex hull [SS97]. Further sections will talk about a few properties of Delaunay Triangulation, briefly describe some of the triangulation approaches, and go through the implementation of the Bowyer-Watson Algorithm.

2 Literature Review

2.1 Properties of Delaunay Triangulation

- **Convex Hull:** The boundary of the triangulation is the convex hull of the set of points being triangulated.
- **Empty Circumcircle:** The circumcircle of any triangle does not contain any other point. This can be understood through the dual voronoi diagram, where each voronoi site is the circumcenter of the triangle, and the three points of the triangle are the nearest neighbours. Therefore we cannot have any other point within the circumcircle.
- **Empty Circle:** Circles with edges of a Delaunay Triangulation do not contain any other point. This can be understood through the dual voronoi diagram too. If **p** and **q** are points with a Delaunay Edge between them, then their voronoi cells are neighbours. For the voronoi edge between these cells, a circle centered at a point **c** on the voronoi edge, passing through **p** and **q**, needs to be empty since **p** and **q** are the nearest neighbours. **c** is equidistant from **p** and **q**, so **c** could also be the midpoint of the Delaunay edge, and therefore the circle with **p** and **q** as the diameter will also be empty.

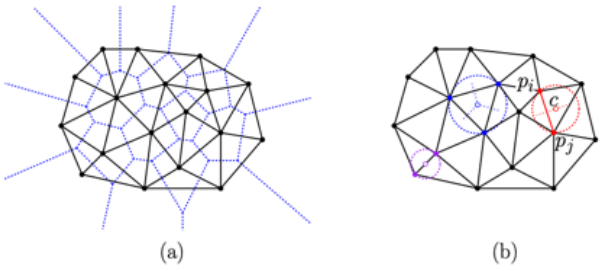
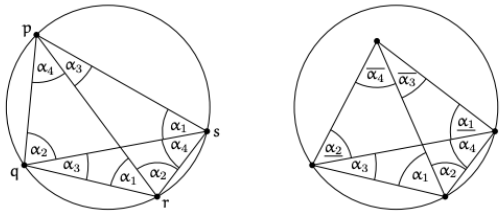


Figure 1: Convex hull and Empty circle [Mou20]

- **Maximum smallest angle:** Delaunay Triangulation maximizes the smallest angle among all triangulations of the set of points. This can be observed by considering the two possible triangulations of a set of points **p,q,r,s**. Initially **p** is on the circumcircle, which is then moved inside. Now the angles $\overline{\alpha_4}$ and $\overline{\alpha_3}$ are strictly larger than α_4 and α_3 according to the inscribed angle theorem. Subsequently $\underline{\alpha_2}$ and $\underline{\alpha_1}$ will be strictly smaller than α_2 and α_1. Listing all angles for both triangulations, you will able to find a smaller angle in the non-Delaunay triangulation, for each angle in the Delaunay triangulation.



(a) Four cocircular points and the induced eight angles.

(b) The situation before a flip.

Figure 2: DT maximizes smallest angle [GH13]

- **Euclidean Minimum Spanning Tree is part of the Delaunay Triangulation:** If you consider the nodes of a graph as points to be triangulated, edges of the euclidean minimum spanning tree will be part of the Delaunay Triangulation. To understand this, consider a minimum spanning tree **T**, with a non-Delaunay edge *ab*. Since this is not a Delaunay edge, from the empty circle property we saw earlier, without losing generality we can say that the circle with *ab* as the diameter could have another point **c** within it. We can replace edge *ab* with a new edge *cb* and get a new tree **T'**. Since *ab* was the diameter, any other line within the circle will be shorter i.e. $len(cb) < len(ab)$. Therefore $weight(\mathbf{T'}) < weight(\mathbf{T})$, which contradicts our initial assumption of **T** being the MST with a non-Delaunay edge.

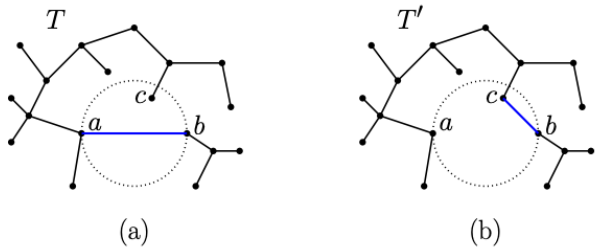


Figure 3: EMST is part of the DT [Mou20]

2.2 Construction of Delaunay Triangulation

Delaunay Triangulation construction algorithms can be broadly classified into five categories [SS97]:

- **Divide and Conquer:** Guibas and Stolfi [GS85] gave an algorithm with O(nlogn) complexity. Dwyer [Dwy87] made a modification to the algorithm, and the approach recursively splits the set of points till you have vertical strips of $\sqrt{n/logn}$ points. Each of these strips is triangulated, and then merged recursively. While merging, the Delaunay Criterion is maintained. This algorithm run in $O(nloglogn)$ time for unformly distributed points.

- **Sweepline:** Fortune [For87] gave an algorithm with O(nlogn) complexity to compute Voronoi diagram. It can be adapted to generate a Delaunay Triangulation instead. The algorithm keeps track of two sets of state - a list of edges called the frontier, and an event queue to check where the sweepline should stop. The sweepline sweeps through the list of points and the frontier is advanced. A site event is added to the event queue when the line reaches a point, and a circle event is added when the line reaches the end of a circle formed by three adjacent vertices of the frontier. And at every such event the Delaunay criterion is enforced.

- **Incremental:** This is the simplest class of algorithms, and as the name suggests it adds points one by one and updates the triangulation incrementally. Most algorithms in this category have two basic

steps, ***locate*** and ***update***. ***locate*** finds the triangle within which the new point lies, and ***update*** modifies the triangulation to conform to the Delaunay criterion. Most algorithms perform the ***update*** in a similar manner, i.e. by flipping edges. The bottleneck of the algorithm is the ***locate*** method, and that's where algorithms tend to differ. Guibas and Stolfi [GS85] propose starting at a random edge, and walking across edges until the correct triangle is found. Every step the check if the point lies on the correct side of the edges of a triangle and decide which edge to step towards. This ***locate*** method is expected to perform $O(\sqrt{n})$ edge tests. Guibas, Knuth and Sharir [GKS92] propose a tree-based approach where the current triangulation is at the leaves, and the internal nodes are triagles that have been deleted or modified. This is expected to work with $O(nlogn)$ complexity. Bowyer [Bow81] and Watson [Wat81] independently proposed a different method that does not use the edge flip method to enforce the Delaunay criterion. Further sections will describe a variant of this algorithm and an implementation.

- **Giftwrapping:** In this class of algorithms we start with a single triangle and incrementally grow the triangulation one at a time, from the edge of the existing triangle. Dwyer [Dwy91] formulated an approach, where an edge (a, b) is taken from a triangle, and a point c is chosen to contruct a triangle. Other points are tested to see which one falls within the circumcircle of the triangle, and if one does, it's chosen to be c. This way the Delaunay criterion is enforced.

- **Convex Hull:** This category of algorithms uses the unique relation between $(d+1)-dimensional$ convex hulls and a $d-dimensional$ Delaunay Triangulation. If 2-D points are lifted onto a 3-D paraboloid $(x, y, x^2 + y^2)$, and the respective convex hull is computed, the 2-D projection of the downward facing faces of this convex hull gives the Delaunay Triangulation. Algorithms such as the Quickhull algorithm [BDH96] can be used to compute the convex hull.

2.3 Performance

Su and Drysdale [SS97] present an experimental comparison of the above mentioned algorithm categories. They found Dwyer's divide an conquer algorithm to be the fastest, with about a factor of 2 in comparison to other techniques on certain machines. It was also the most resistant to bad point distribution with an $O(nlogn)$ worst case. Barber's Quickhull algorithm was much slower than the other algorithms analysed. Since it is actually an algorithm to compute convex hulls in a stable way, it uses different primitives and has a lot more overhead than needed to cmopute 2-D Delaunay triangulations. Fortunes sweepline algorithm along with bucketing the points close to each other was found to be the second fastest, which in comparison to Dwyer's algorithm, was around 50% slower on uniform datasets, and up to 85% slower on non-uniform datasets. It was also found that for large datasets with 2^{14} points and more, Fortune's sweepline algorithm and incremental algorithms had similar performance.

3 Implementation

3.1 Bowyer-Watson Algorithm

The Bowyer-Watson Algorithm is an incremental algorithm, which means that the algorithm iteratively adds points to a triangulation, and at the end of each iteration, modifies the triangulation such that the Delaunay criterion is maintained. A relatively simple variant of this algorithm was described by Paul Bourke [Bur89], which is shown below.

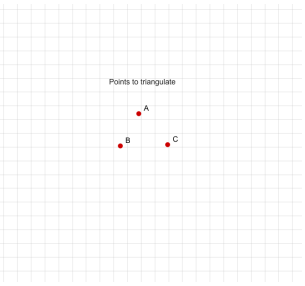
The algorithm needs an initial triangulation to begin with. Therefore, the first step is to create a large triangle which will contain all the points that need to be triangulated within its edges. This is called the 'super triangle'. There are multiple ways to generate the super-triangle, such as computing a minimum bounding triangle, or computing a minimum enclosing circle and then an enclosing triangle. But for our purposes a simpler approach suffices, where we compute the rectangular bounding box of all the points and center a triangle around this rectangle.

The next step is to add the first point to the triangulation. We check if the point falls within the circumcircle of any existing triangles, and mark those triangles as invalid. Further, we delete these triangles, which will leave us with a polygonal hole which needs to be filled. This is done by connecting the edges of the polygonal hole with the newly added point. This generates a new set of triangles and the newly added point lies on the circumcircle of these triangles and hence does not invalidate the Delaunay criterion.

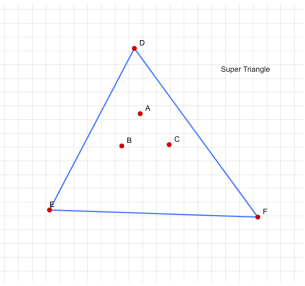
Finding the polygonal hole after marking invalid triangles can be tricky. The hole is defined by the boundary formed by all the invalid triangles. To get the boundary we need to delete all the internal edges,

which are the edges shared by any two invalid triangles. Therefore the polygonal hole is given by the edges which appear only once in the list of invalid triangles.

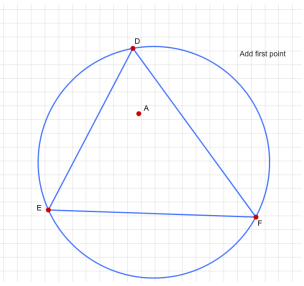
The steps above are repeated for all the points that need to be triangulated. Finally we need to get rid of the super-triangle that was introduced in the first step. This is done by iterating through all the triangles in our triangulation and deleting the ones which share a vertex with the super triangle. In the end we are left with a valid Delaunay triangulation of the initial set of points. The figure below visually illustrates the algorithm.



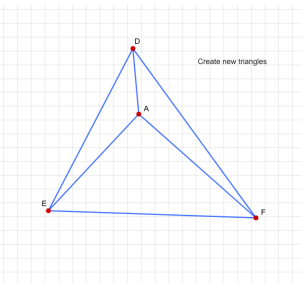
(a) Points



(b) Super-triangle

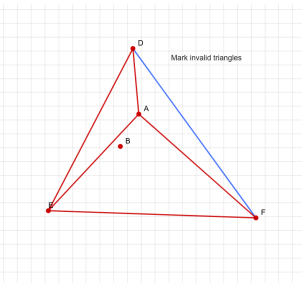


(c) Add first point

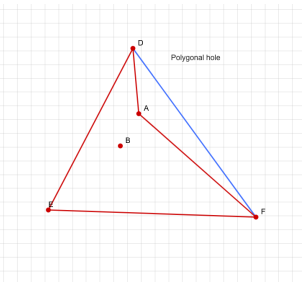


(d) Generate new tris

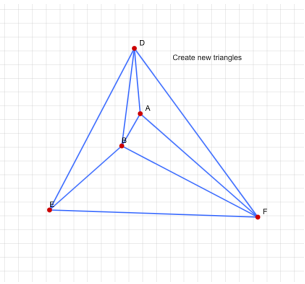
(e) Add second point



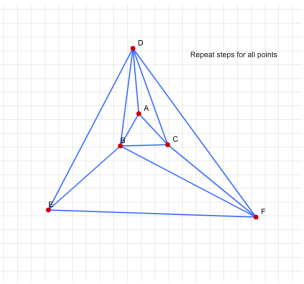
(f) Mark invalid tris



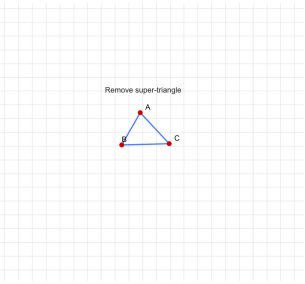
(g) Polygonal hole



(h) Generate new tris



(i) Repeat for all points



(j) Remove super-triangle

Figure 4: Bowyer-Watson Algorithm visual example

Algorithm 1: Bowyer-Watson Algorithm to generate a Delaunay Triangulation from a list of points

```
1  function Triangulate (points[]);
   Input  : A list of 2D points (points[])
   Output: A list of triangles
2  tris[] = {};
3  super-triangle[] = getSupertriangle(points);
4  tris.add(super − triangle);
5  for i ← 0 to points.size() by 1 do
6      edges[] = {};
7      uniqueEdges[] = {};
8      newTris[] = {};
9      vertex = points[i];
10     for j ← 0 to tris.size() by 1 do
11         t = tris[j];
12         if inCircumcircle(t, vertex) then
13             edges.add(Edge(t.v0, t.v1));
14             edges.add(Edge(t.v1, t.v2));
15             edges.add(Edge(t.v2, t.v0));
16         else
17             newTris.add(t);
18         end
19     end
20     tris = newTris;
21     for j ← 0 to edges.size() by 1 do
22         isUnique = true;
23         for k ← 0 to edges.size() by 1 do
24             if j != k && edges[j] == edges[k] then
25                 isUnique = false;
26                 break;
27             end
28         end
29         if isUnique then
30             uniqueEdges.add(edges[j]);
31         end
32     end
33     for j ← 0 to uniqueEdges.size() by 1 do
34         e = uniqueEdges[j];
35         tris.add(Triangle(e.v0, e.v1, vertex));
36     end
37 end
```

3.2 Computing the Voronoi Diagram

We can compute the Voronoi Diagram from an existing Delaunay Triangulation by connecting the circumcenters of neighbouring triangles. In the implementation shown above, we do not store the adjacency data. We just store the list of points, and and a list of triangles with indices referring to this list of points. But we use the edge primitive extensively in the algorithm. If we store the circumcenter along with each edge of the triangle, in the end each edge will have at most two linked circumcenters. Therefore, at the end of the triangulation we create a list of edges, along linked to a circumcenter. There might be duplicate edges, each linked to a different circumcenter. Now this list needs to be collated to find unique edges mapped to a list of circumcenters. If this is done, we have found all our Voronoi edges (line between the two circmcenters mapped to the Delaunay edge). In order to do this, we use a Hash Map, along with a custom hash function for the Edge datastructure. The custom hash function sorts the vertex indices of the edge, hashes them (using the hash struct in the C++ standard library), and combines them using the way the boost library combines hashes.

Algorithm 2: Computing the Voronoi edges from an existing Delaunay Triangulation

```
function ComputeVoronoi ();
Input  : A list of triangles (tris[])
Output: A list of voronoi edges
voronoiEdges = map < Edge, List < Point >>;
for i ← 0 to tris.size() by 1 do
    tris[i].populateEdgesWithCircumcenter();
end
for i ← 0 to tris.size() by 1 do
    edges[] = tris[i].getEdges();
    voronoiEdges[edges[0]].add(edges[0].circumCenter);
    voronoiEdges[edges[1]].add(edges[1].circumCenter);
    voronoiEdges[edges[2]].add(edges[2].circumCenter);
end
return voronoiEdges;
```

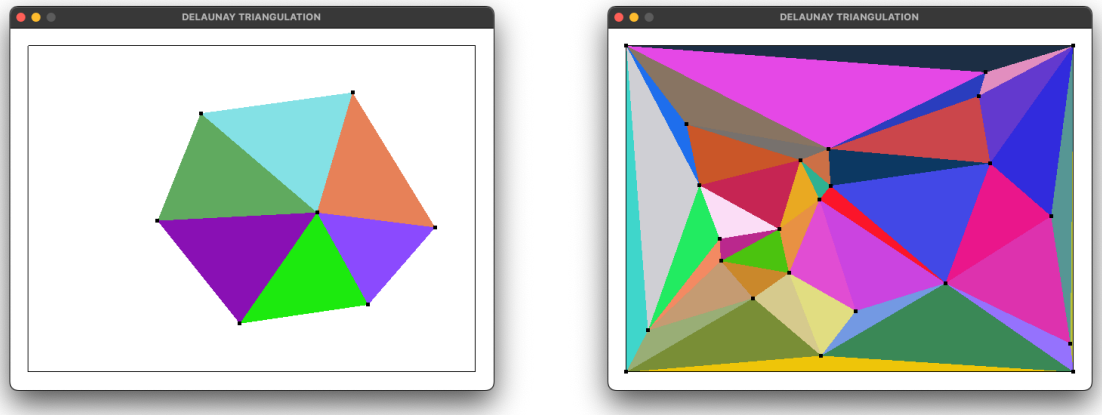
3.3 Testing

The GUI application has the following features:

- Generate a random set of points by pressing the ‘G‘ key.
- Triangulate the set of points by pressing the ‘T‘ key.
- Add custom points by left-clicking within the domain.
- Clear the points by pressing the ‘C‘ .key.

The following tests were performed to verify the implementation:

- **Triangulate a random set of points**: Triangulating randomly distributed point seems to work correctly. The boundary of the triangulation is the convex hull of the point set, as expected. For a set of 100 points, the program takes around 1031 microseconds on average to compute the Delaunay triangulation and the Voronoi diagram, where the ratio of the time taken for triangulation and the time taken to compute the Voronoi diagram is around 6:1.



(a) Triangulate custom points (b) Triangulate a random set of points

Figure 5: Bowyer-Watson Algorithm visual example

- **Triangulate a dataset with collinear points:** With a set of 7 points where the three left most points are collinear, the triangulation works correctly.

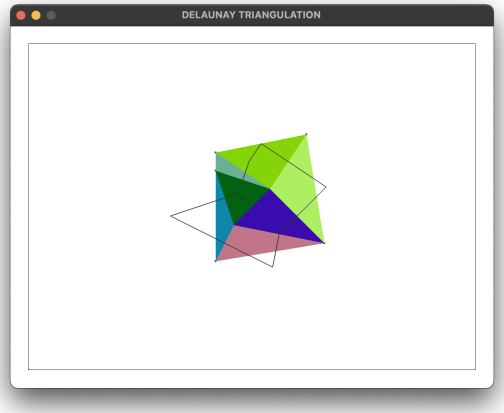
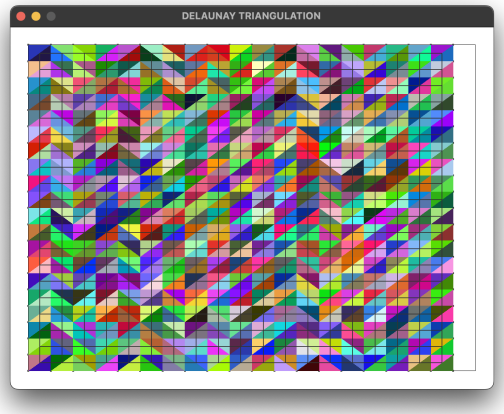
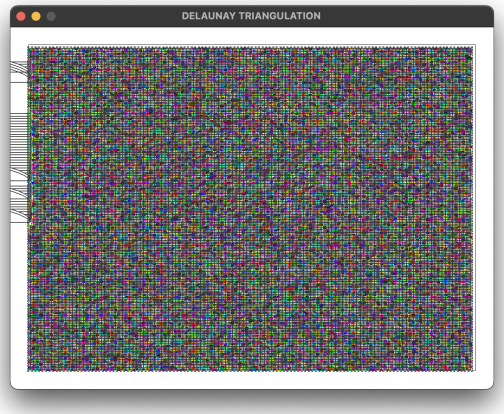


Figure 6: Triangulating a pointset with collinear points

- **Triangulate a rectangular grid, which consists entirely of co-circular points:** Triangulating a rectangular grid works for grid sizes of up to 62x62, after which it breaks, the reason for which I have not been able to pinpoint yet.



(a) 20x20 grid



(b) 63x63 grid

Figure 7: Triangulating a grid of co-circular points

4 Conclusion

The properties of Delaunay Triangulation, specifically that it avoids slivers makes it a very lucrative triangulation amongst the many possible, for wide ranging applications including terrain modelling, meshing for FEM, path planning, point cloud density estimation and generative art. Most Delaunay Triangulation algorithms can be implemented in O(nlogn) time complexity, but Dwyer's divide and conquer turns out to be the strongest across wide range of datasets, with uniform and non-uniform distribution. For larger datasets sweepline and incremental algorithms perform similarly. Although an interesting observation, convex hulls algorithms for Delaunay triangulation are much slower, and infeasible for practical purposes. The Bowyer-Watson incremental algorithm presented in the report is an $O(n^2)$ implementation. This makes the implementation overhead minimal and is a good approach to develop an intuition of the algorithm and understand how the Delaunay criterion can be enforced practically. The implementation can be used for use-cases with smaller datasets like those used for algorithmic art, but it can still be unreliable at times. A lot of work has been done to research various algorithms and to create standard implementations in packages such as CGAL and MATLAB. These implementations should be used for scientific use-cases or with larger datasets.

References

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, dec 1996.

[Bow81] A. Bowyer. Computing Dirichlet tessellations*. *The Computer Journal*, 24(2):162–166, 01 1981.

[Bur89] Paul Burke. Efficient triangulation algorithm suitable for terrain modelling. *Pan Pacific Computer Conference*, 1989.

[Dwy87] Rex A. Dwyer. A faster divide-and-conquer algorithm for constructing delaunay triangulations. *Algorithmica*, 2(1):137–151, Nov 1987.

[Dwy91] Rex A. Dwyer. Higher-dimensional voronoi diagrams in linear expected time. *Discrete & Computational Geometry*, 6(3):343–367, Sep 1991.

[For87] Steven Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2(1):153–174, Nov 1987.

[GH13] Bernd Gärtner and Michael Hoffmann. CS268 Computational Geometry Lecture notes. 2013.

[GKS92] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica*, 7(1):381–413, Jun 1992.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4(2):74–123, apr 1985.

[Mou20] Dave Mount. CMSC 754 Computational Geometry Lecture notes. 2020.

[SS97] Peter Su and Robert L. Scot Drysdale. A comparison of sequential delaunay triangulation algorithms. *Computational Geometry*, 7(5):361–385, 1997. 11th ACM Symposium on Computational Geometry.

[Wat81] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes*. *The Computer Journal*, 24(2):167–172, 01 1981.