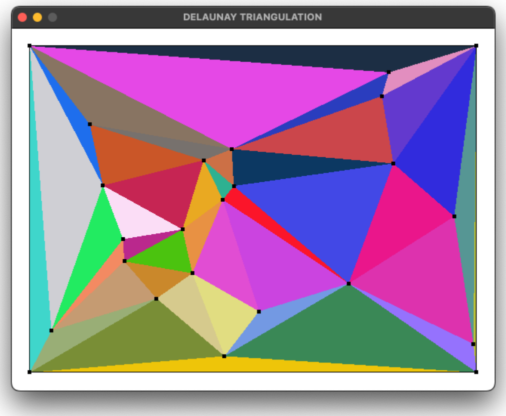
Delaunay Triangulation and Voronoi Diagram

Karthik Iyer

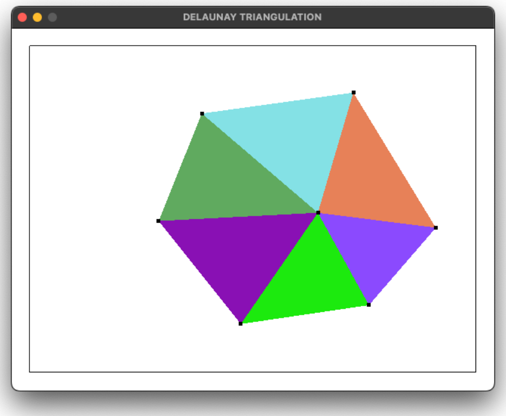
1 Progress Update

I have implemented a program that uses the Bowyer-Watson Algorithm to triangulate a set of points in the 2D plane and uses OpenGL for visualization. The GUI program allows the user to:

- 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.



(a) Triangulate Random Points



(b) Triangulate Custom Points

Figure 1: GUI Program

2 Bowyer-Watson Algorithm

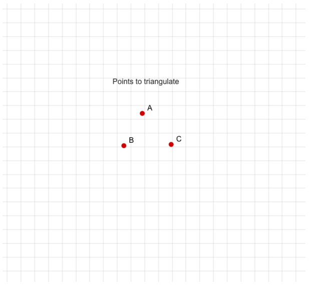
The Bowyer-Watson Algorithm is an incremental algorithm, which means that the algorithm iteratively adds points to a triangulation, and at the end of each iteration, modifies the triangulation such that the Delaunay criterion is maintained. A relatively simple variant of this algorithm was described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

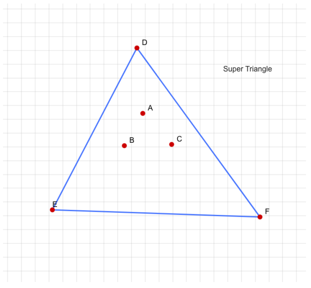
The next step is to add the first point to the triangulation. We check if the point falls within the circumcircle of any existing triangles, and mark those triangles as invalid. Further, we delete these triangles, which will leave us with a polygonal hole which needs to be filled. This is done by connecting 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.

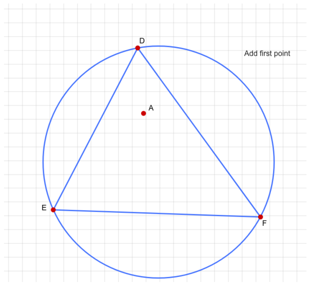
The steps above are repeated for all the points that need to be triangulated. Finally we need to get rid of the super-triangle that was introduced in the first step. This is done by iterating through all the triangles in our triangulation and deleting the ones which share a vertex with the super triangle.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



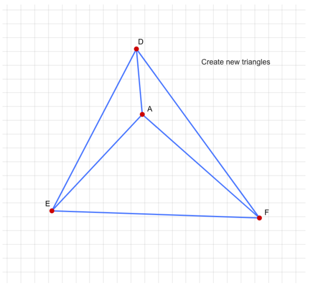
(a) Points



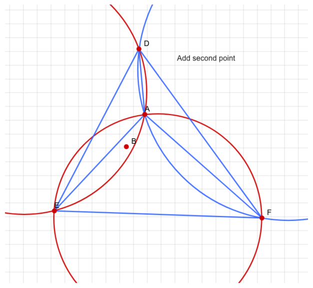
(b) Super-triangle



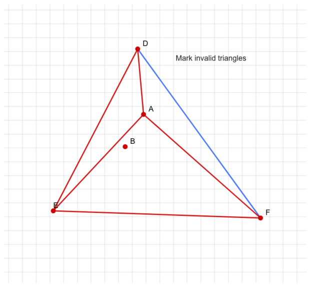
(c) Add first point



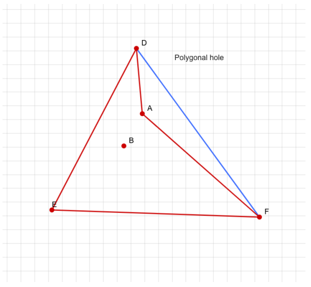
(d) Generate new tris



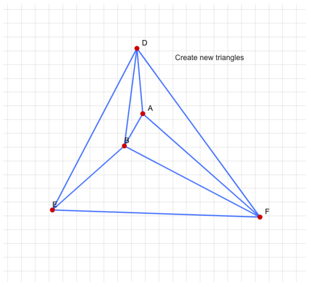
(e) Add second point



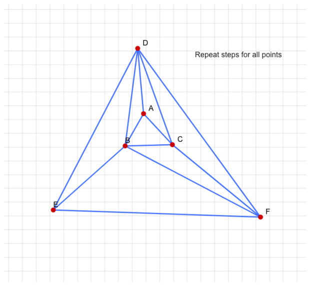
(f) Mark invalid tris



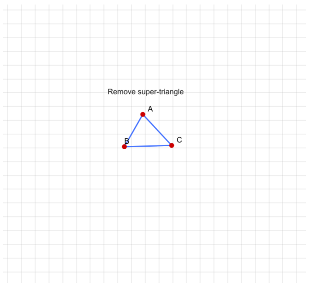
(g) Polygonal hole



(h) Generate new tris



(i) Repeat for all points



(j) Remove super-triangle

Figure 2: Bowyer-Watson Algorithm visual example

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

```
function Triangulate ($points$[]);
Input  : A list of 2D points ($points$[])
Output: A list of triangles
tris[] = {};
super-triangle[] = getSupertriangle($points$);
tris.add($super - triangle$);
for $i \leftarrow 0$ to $points.size()$ by 1 do
    edges[] = {};
    uniqueEdges[] = {};
    newTris[] = {};
    vertex = points[$i$];
    for $j \leftarrow 0$ to $tris.size()$ by 1 do
        t = tris[$j$];
        if $inCircumcircle(t, vertex)$ then
            edges.add($Edge(t.v0, t.v1)$);
            edges.add($Edge(t.v1, t.v2)$);
            edges.add($Edge(t.v2, t.v0)$);
        else
            newTris.add(t);
        end
    end
    tris = newTris;
    for $j \leftarrow 0$ to $edges.size()$ by 1 do
        isUnique = $true$;
        for $k \leftarrow 0$ to $edges.size()$ by 1 do
            if j != k && edges[j] == edges[k] then
                isUnique = $false$;
                break;
            end
        end
        if $isUnique$ then
            uniqueEdges.add($edges[j]$);
        end
    end
    for $j \leftarrow 0$ to $uniqueEdges.size()$ by 1 do
        e = uniqueEdges[$j$];
        tris.add($Triangle(e.v0, e.v1, vertex)$);
    end
end
```

3 Pending goals

- Extend the triangulation implementation to find the dual Voronoi digram by connecting the centroid of the triangles.
- Compute the Centroidal Voronoi Tesselation (CVT) based on Lloyd's relaxation algorithm.

References

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