Creating Voronoi Diagrams Using Delaunay Tetrahedralisations

Trinity College Dublin

Seán Martin

March 20, 2017

Abstract

A method for computing and visually displaying the Voronoi diagram and Delaunay triangulation of point sets is presented in two and three dimensions, and supplemented by C code. Firstly the Bowyer Watson algorithm is implemented to produce the Delaunay triangulation and the Voronoi diagram is extracted from this. This project begins by defining the relevant geometrical notions, then moves on to discussion of the algorithms and data structures that are used, ending with results, applications and C code.

Declaration

This thesis is my own work except where due citations are given. I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.
I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

Acknowledgements

Many thanks to my supervisor, Prof. Colm Ó Dúnlaing for all the help and guidance with this project, I'm not sure where I would have even started without him. A former TCD student, Andrew Farrell produced a thesis with Prof. Ó Dúnlaing which I found very useful, but ultimately does not appear anywhere in this document [Far95]. Finally, thanks to family and friends for dealing with exclamations of "Why won't this work!" and "I just don't understand!".

Contents

Introduction

Delaunay triangulations and Voronoi diagrams were theorised in reverse order to how we shall use them. The origin of the Voronoi diagram dates back to the 17th century with illustrations in René Descartes' *Principia philosophiae* resembling the modern Voronoi Diagram [Des44] (See pg. 78). The following historical account is from Klein and Aurenhammer [AK00]. They explain how Descartes' illustrations show a decomposition of space into convex regions, each consisting of matter revolving around one of the fixed stars.

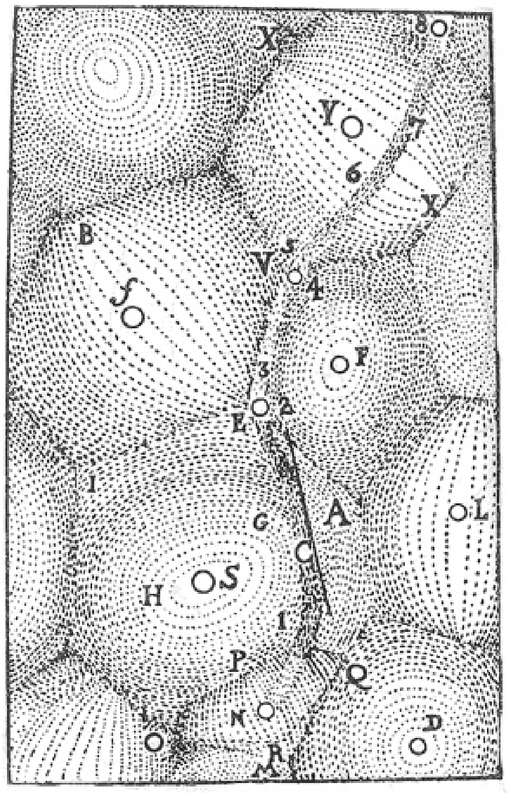


Figure 1: Extracted from [Des44]. Descartes' decomposition of space - for example the bounded convex polygon surrounding the vertex f resembles a Voronoi region.

Although Descartes did not formally define the Voronoi diagram, he certainly lay the foundations for Dirichlet and Voronoi to formally introduce the concept. Hence the alternative name Dirichlet tesselation. Voronoi himself was the first to consider the dual of the Voronoi Diagram, where any two points whose Voronoi regions have boundary in common are connected by an edge. Delaunay later defined the Delaunay triangulation, and found it to be the same as the dual of the Voronoi Diagram. We will be going in the opposite direction; using the Delaunay triangulation to compute its dual, the Voronoi Diagram.

Both the Voronoi diagram and Delaunay triangulation of point sets have widespread applications (as do more general versions, which deal with convex 'sites' - instead of points [HÓDY07], or perhaps have fuzzy edges). For example, the Voronoi diagram has applications in natural growth models, city planning, organic texture generation, and geostatistics. Furthermore, much research has been devoted to their study, and much literature on the topic is very recent. For those who would be interested in further reading, or perhaps improving this implementation, I would highly recommend referring to [LS05] which breaks down five different implementations of 3D Delaunay triangulation. Another option is somewhat older but, [GS85] presents a very interesting Quad-Edge data structure, and computes Delaunay triangulations using subdivisions of manifolds.

Before we begin, let us make one note about terminology. We shall use the term *Delaunay triangulation* as a general term for cases with points in $\mathbb{R}^d$ and the term *Delaunay tetrahedralisation* to refer spefically to the case with points in $\mathbb{R}^3$. Generally speaking, the 2D case will be discussed as it is easier to visualise and is readily adapted to 3D.

The general structure of this project will be as follows. We define the Delaunay triangulation and Voronoi diagram and how they are dual. Next we move on to the Bowyer Watson algorithm, its correctness and how the Voronoi diagram is extracted from this. Finally we discuss improving the efficiency of the implementation, shortcomings of the implementation and results.

1 Definitions

1.1 Triangulations of Point Sets

Most definitions will be given in d dimensions, even though the code will deal with only two and three dimensions. This is because the Boywer Watson algorithm works in higher than three dimensions, and so the Voronoi diagram can be extracted as the dual in higher than three dimensions. Furthermore, we avoid having to define terms in two and three dimensions separately.

Definition 1.1. The *convex hull* of a point set $A \subset \mathbb{R}^d$ is the smallest convex set that contains A, and is denoted $H(A)$. A *convex set* in $\mathbb{R}^d$ is a set of points such that given any pair of points in the set, the straight line segment joining the pair of points is fully contained in the set. See Figure 2.

Definition 1.2. Points $x_1, \dots, x_n \in \mathbb{R}^d$ are *affinely independent* if any linear combination $\lambda_1 x_1 + \cdots + \lambda_n x_n = 0$ with $\lambda_1 + \cdots + \lambda_n = 0$ must have $\lambda_1 = \cdots = \lambda_n = 0$.

Definition 1.3. A *k-simplex* is the convex hull of $k+1$ affinely independent points in $\mathbb{R}^d$. These points are referred to as the *vertices* of the simplex.

From the definitions, $\mathbb{R}^d$ can contain at most a d-simplex. When we create programs, we shall be dealing with $\mathbb{R}^2$ and $\mathbb{R}^3$ so the biggest simplex we shall see is a 3-simplex, namely a tetrahedron.

Definition 1.4. Let σ and τ be simplices in $\mathbb{R}^d$, with vertices A and B respectively. Then we say that τ is a *face* of σ if $B \subseteq A$. If τ is a k-simplex we say it is a k-face of σ.

Definition 1.5. A finite collection K of simplices in $\mathbb{R}^d$ is said to be a *simplicial complex* if the following two conditions are satisfied:

1. If σ belongs to K then every face of σ belongs to K.

2. If σ and τ belong to K then either σ and τ are disjoint faces, or they intersect along a common face of σ and τ.

If the biggest simplex in K is a k-simplex, then K is said to be a simplicial k-complex.

Definition 1.6. We shall use a slightly modified definition from De Loera et Al [LRS10]. A *triangulation* of a point set $A \in \mathbb{R}^d$ is a simplicial d-complex K with vertices A such that the union of all d-simplices in K is $H(A)$. See Figure 3.

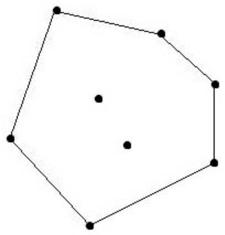


Figure 2: Convex Hull of 8 points

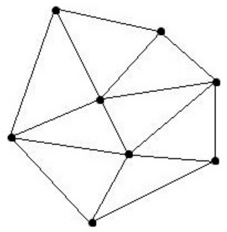


Figure 3: Triangulation of 8 points

1.2 Convex Polytopes

The following definitions are from Matoušek [Mat02].

Definition 1.7. A *hyperplane* in $\mathbb{R}^d$ is a set of the form $\{x \in \mathbb{R}^d : \ a \cdot x = b\}$ with $a \in \mathbb{R}^d$, $a \neq 0$ and $b \in \mathbb{R}$. A *closed half-space* in $\mathbb{R}^d$ is a set of the form $\{x \in \mathbb{R}^d : \ a \cdot x \geq b\}$, with a and b as above. Note that $x \cdot y$ denotes the regular algebraic dot product, $x_1y_1 + x_2y_2 + \cdots + x_ny_n$ for $x, y \in \mathbb{R}^n$.

Remark. From the above definitions, it is clear that a hyperplane forms the boundary of a closed half-space. Thus a hyperplane in $\mathbb{R}^d$ determines two closed half-spaces and the union of these half-spaces is $\mathbb{R}^d$.

Definition 1.8. A *convex polytope* P in $\mathbb{R}^d$ is the intersection of finitely many closed half-spaces in $\mathbb{R}^d$. P is called an n-polytope if it has affine dimension n.

Definition 1.9. A *face* of a convex d-polytope P is defined as either

1. P itself.
2. $P \cap h \neq \varnothing$, where h is a hyperplane such that P is fully contained in one of the closed half-spaces determined by h.

A face will be a convex polytope. If a face of P is an n-polytope, then it is called an n-face of P. By definition P has faces of dimension $0, 1, \ldots, d$ where 0-faces are vertices.

As an example, a regular hexagon is a bounded convex 2-polytope (or simply a polygon) with

- One 2-face, the hexagon itself.
- Six 1-faces, the edges of the hexagon.
- Six 0-faces, the vertices of the hexagon.

1.3 Delaunay Triangulations

Some authors would take the following to be a theorem, for example [DO11], but we shall take it to be a definition:

Definition 1.10. A *Delaunay triangulation* $DT(A)$ of a point set $A \subset \mathbb{R}^2$ is a triangulation of A such that no point of A lies inside the circumcircle of a triangle in $DT(A)$. A triangle having no points in the interior of its circumcircle is often referred to as having the *empty circle property.*

We can generalise the above definition to a point set $A \subset \mathbb{R}^d$. But first we need the notion of a d-dimensional disc, or d-disc.

Definition 1.11. A d-dimensional open disc, or simply an *open d-disc*, of radius r and centre c is the set of points

$$\{x \in \mathbb{R}^d : d(x,c) < r\} \text{ where } d(x,c) \text{ is the Euclidean distance between } x \text{ and } c.$$

Remark. Since d-simplices consist of $d+1$ vertices, the vertices of a d-simplex σ define an open d-disc such that the boundary of the disc passes through all the vertices of σ. The boundary of this open d-disc is a $(d-1)$-hypersphere which circumscribes σ.

Definition 1.12. A *Delaunay triangulation* $DT(A)$ of a point set $A \subset \mathbb{R}^d$ is a triangulation of A such that no point of A lies in an open d-disc whose boundary circumscribes a d-simplex in $DT(A)$.

1.4 Voronoi Diagrams

We will use a slightly modified and generalised definition from Liebling and Pournin [LP12].

Definition 1.13. Consider a point set $A \subset \mathbb{R}^d$. The *Voronoi region R_x* associated with the point x in A is a possibly unbounded convex d-polytope which consists of those points in $\mathbb{R}^d$ whose distance to x is not greater than their distance to any other point of A.

Definition 1.14. The *Voronoi diagram $V(A)$* induced by A is a decomposition of $\mathbb{R}^d$ into the Voronoi regions associated with the points of A. $V(A)$ will often be referred to as the Voronoi diagram of A.

Remark. Notice that a Delaunay triangulation of a point set A is not always unique, while the Voronoi diagram of a point set A is unique.

1.5 Describing Duality

A vertex in the Voronoi diagram or Delaunay triangulation will be called a Voronoi vertex or Delaunay vertex, respectively, and similarly for other geometrical structures. Given a point set $A \subset \mathbb{R}^d$ the duality between $V(A)$ and $DT(A)$ is the following. Each n-face of a Voronoi d-polytope (Voronoi region) corresponds to one and only one $d-n$ face of a Delaunay d-simplex (Delaunay triangle). We will explicitly describe this duality in two and three dimensions.

In two dimensions, each Voronoi vertex corresponds to a Delaunay triangle, each Voronoi edge corresponds to a Delaunay edge, and each Voronoi polygon corresponds to a Delaunay vertex. An edge in $V(A)$ is always a line segment or a ray of the perpendicular bisector of its corresponding Delaunay edge.

In three dimsensions there is an informative picture from [Led07] which demonstrates the situation well. Figure 4 shows, in order:

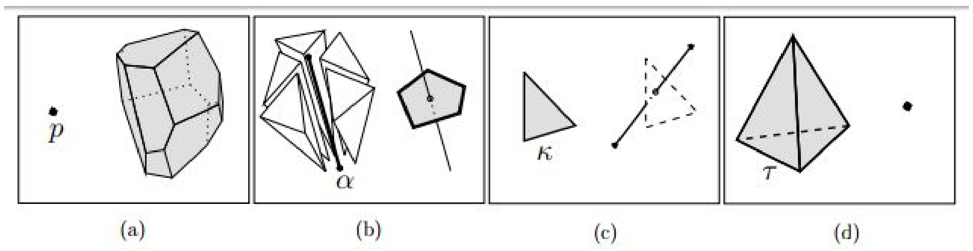


Figure 4: From [Led07], demonstrating the duality in 3D between $V(A)$ and $DT(A)$.

(a) A Delaunay vertex p corresponds to a Voronoi Polyhedron (Voronoi region).

(b) A Delaunay edge α corresponds to a Voronoi face.

(c) A Delaunay face κ corresponds to a Voronoi edge.

(d) A Delaunay tetrahedron τ corresponds to a Voronoi vertex.

2 Bowyer Watson Algorithm

There are many algorithms to compute the Voronoi diagram directly for a point set in 2D, and many algorithms which first compute the Delaunay triangulation and then the Voronoi diagram from this in 2D. However many of these algorithms do not generalise to higher dimensions. As such we will follow the algorithm presented independently by both Bowyer and Watson, which functions in n-dimensions [Wat81] [Bow81].

2.1 Algorithm Description

The Bowyer Watson algorithm gives an incremental method of producing the Delaunay triangulation of point sets $A \subset \mathbb{R}^n$. To quote [LS05], which compared five Delaunay tetrahedrilisation programs, "Each of the five programs compute the Delaunay tessellation incrementally, adding one point at a time". I believe incremental construction of Delaunay triangulations and tetrahedrilisations is commonplace because it is fast and easy to understand. We shall give a very basic implementation of the Boywer Watson algorithm in pseudo code, and later discuss how we develop a more sophisticated implementation. The pseudo code given here will be for the 2D case, but it generalises readily.

Remark. In the following sections we will call triangles 'bad', or 'good'. A triangle becomes bad if it no longer satisfies the empty circle property after a new point of A is added, and good otherwise.

```
//Data: Input point set A, two empty sets of triangles Del and Bad and a polygon P
//Result: Del will be a Delaunay triangulation of A
Triangle_set bowyer_watson(Point_set A) {
  Polygon P; Triangle_set Bad;
  create a super triangle which contains all points of A and add it to Del;
  for (each point x in A) {
    empty Bad; //Clear the set of bad triangles
    for (each triangle T in Del) { //Find the new bad triangles
      if (x lies inside the circumcircle of T) {
          add T to Bad and remove T from Del;
      }
    }
    for (each triangle T in Bad) { //Find the boundary of the bad triangles
      if (an edge of T is not shared by another triangle in Bad) {
        add that edge to P;
       }
    }
    for (each edge e of P) { //Retriangulate inside P
      form a new triangle by joining e to x and add this triangle to Del;
    }
  }
  Remove all triangles which share a vertex with the super triangle from Del;
  return Del;
}
```

Remark. To generalise this algorithm from two dimensions to n-dimensions, replace all instances of the word triangle with n-simplex, edge with $n-1$ face, the polygonal convex set P with an n-polytope and the circumcircle of a triangle with the $(n-1)$-hypersphere circumscribing an n-simplex. So in three dimensions we have triangles replaced by tetrahedrons, edges replaced by polygons, polygonal convex sets replaced by polyhedrons, and the circumcircle of a triangle replaced by the circumsphere of a tetrahedron.

In Figure 5 we present an example of the Bowyer Watson algorithm simulated on five points in $\mathbb{R}^2$. At each step after (a), the new Delaunay edges are shown in blue. Notice that at each step, we add at most two triangles to the triangulation, which is a general result in two dimensions.

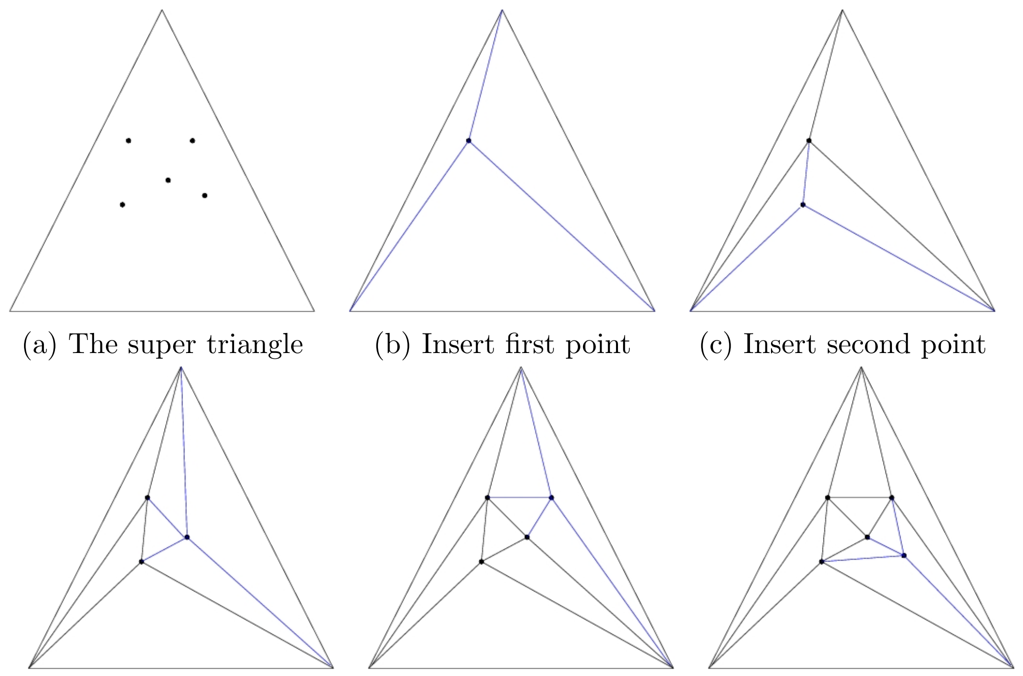
Removing all the triangles from the triangulation which share a vertex with the original 'super' triangle gives the Delaunay triangulation of the five points (Figure 6).

2.2 Degeneracies

Let $A \subset \mathbb{R}^2$. $DT(A)$ does not exist if all points of A are collinear. If four or more points in A lie on the same circle, then $DT(A)$ will not be unique. We shall not concern ourselves with the first case, and so we move on to notion of general position. In the second case, it will be a goal of ours to show that we still get the correct Voronoi Diagram no matter which Delaunay triangulation of A we choose.

Definition 2.1. Every author's definition of *general position* will change depending on their needs. In our case we say that a point set $A \subset \mathbb{R}^d$ is not in general position if two points in A are non-distinct, or if all points in A lie on a hyperplane in $\mathbb{R}^d$. That is, in 2D and 3D, collinear and coplanar point sets are respectively not in general postion.

Theorem 2.1. *Given a point set $A \subset \mathbb{R}^d$, with $|A| > d+1$ lying on a $(d-1)$-hypersphere, a Delaunay triangulation of A is non-unique. However, each of these Delaunay triangulations produces the same Voronoi Diagram as their dual.*


(a) The super triangle (b) Insert first point (c) Insert second point

(d) Insert third point (e) Insert fourth point (f) Insert fifth point

Figure 5: Every step of the Boywer Watson algorithm in our example, bar the last.

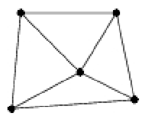


Figure 6: Delaunay triangulation of the five points.

Proof. Since every d-simplex in a Delaunay triangulation $DT(A)$ shares the same circumcentre, we have no Voronoi edges between any simplices in $DT(A)$. Thus every Voronoi edge is defined by the shared circumcentre of the d-simplices and all the $d-1$ faces of $H(A)$. However, any choice of a Delaunay triangulation of A covers $H(A)$ by definition. Thus each Delaunay triangulation produces the same Voronoi edges, and thus the same Voronoi diagram. $\square$

2.3 Correctness

We shall prove the correctness of the Bowyer Watson algorithm for a point set $A \subset \mathbb{R}^2$. The argument for higher dimensions is given by Watson in [Wat81], but it is nice to visualise the two dimensional case.

Definition 2.2. Two distinct triangles are called *neighbours* if they share an edge. A set of triangles C is said to be *strongly connected* if $|C| = 1$, or given any two distinct triangles $T_a, T_b \in C$ we can find a sequence $\sigma_1, \sigma_2, \ldots, \sigma_k$ of triangles in C with $T_a = \sigma_1, T_b = \sigma_k$ where σ_i and σ_{i+1} are neighbours for $i = 1, 2, \ldots, k-1$ and *disconnected* otherwise. A strongly connected set of triangles C is said to *connect* triangle T_a to triangle T_b if T_a and T_b are not in C, but both have neighbours in C.

Definition 2.3. A point x lies in the interior of a Delaunay triangulation of A if x lies inside $H(A)$, the convex hull of A.

Definition 2.4. Consider a Delaunay triangulation of A, $DT(A)$. The *Delaunay cavity* $DC(x)$ of a point x is the set of bad triangles formed by inserting x into $DT(A)$. See Figure 7.

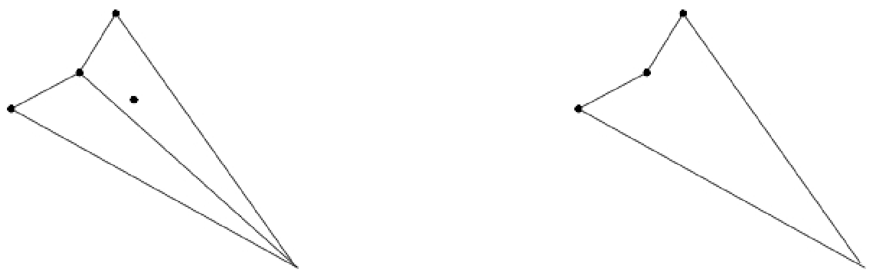

(a) The Delaunay cavity (b) The boundary of the cavity

Figure 7: The Delaunay cavity and the boundary of the cavity, for the last point in the example covered in Figure 5, subfigure 5f.

Lemma 2.2. *Given $DT(A)$, the Delaunay cavity $DC(x)$ of a point x which lies in the interior of $DT(A)$ is always strongly connected and non-empty.*

Proof. Choose a triangle T in $DT(A)$ such that x is contained inside T, or on the border of T. We are guaranteed that T exists becuase we insert x into the interior of $DT(A)$. T is then a bad triangle, and so $DC(x)$ is non-empty as required. Assume that there exists some triangle $B \in DC(x)$ that is not a neighbour of T and is impossible to connect to T using triangles from $DC(x)$. Then any $C \subset DT(A)$ which connects B to T must contain a triangle $T'_C \notin DC(x)$. Then the circumcircle of B must contain a vertex belonging to at least one of these triangles T'_C that is not a vertex of B, otherwise it could not contain x. Thus we did not start with a Delaunay triangulation, a contradiction. □

Definition 2.5. A set $S \subset \mathbb{R}^n$ is called *star-shaped* if there exists $p \in S$ such that the line segment px lies in S for all $x \in S$. The *kernel* of a star-shaped set S is the set $\{p \in S : px \subset S, \forall x \in S\}$.

Lemma 2.3. *Given $DT(A)$, the union of the triangles in the Delaunay cavity $DC(x)$ of a point x which lies in the interior of $DT(A)$ forms a star-shaped set with x in its kernel.*

Most papers on the topic of vertex deletion from Delaunay triangulations state that the Delaunay cavity is star-shaped, but very few papers actually prove this. See [Buc+13] for example, which also gives different methods of re-triangulating the Delaunay cavity than the one presented in this paper. The result makes sense, but unfortunately we shall be no different and will not provide a proof. The idea is that there is at least one triangle T in $DC(x)$ such that T is star-shaped with x in its kernel. Then any other triangle in $DC(x)$ is fully reachable from $x \in T$ by line segments that pass through triangles in $DC(x)$ which are connected to T so the union of triangles in $DC(x)$ is star-shaped. This is as every triangle in $DC(x)$ must contain x in its circumcircle and no other points of A. The basis of our proof of correctness of the Bowyer Watson algorithm comes from Watson [Wat81]:

Theorem 2.4. *Given a correct Delaunay triangulation $DT(A)$ for a set A of $n-1$ points, adding one point x to the interior of $DT(A)$ and retriangulating as outlined in Bowyer Watson algorithm gives a correct Delaunay triangulation of the n points.*

Proof. By the above lemma, the Delaunay cavity formed when adding x into $DT(A)$ is strongly connected, non-empty and the union of the triangles in the cavity form a star-shaped set. As such we can find the polygonal boundary of $DC(x)$. Only the edges that form this polygonal boundary will form the new triangles with x as any other triangle formed inside the boundary would overlap with a triangle formed by the edges on the boundary. Thus the retriangulation does form a triangulation of the n points, we must verify that it is still Delaunay.
By an argument of symmetry, if any of these new triangles were to contain a point of A in its circumcircle, then there would also be some triangle in $DT(A)$ that has not been removed containing x in its circumcircle. This is impossible as we begin by removing all triangles from $DT(A)$ which contain x in their circumcircle. Thus every triangle in the triangulation of the n points satisfies the empty circle property, and we thus obtain a Delaunay triangulation of the n points. □

Theorem 2.5. *The Bowyer Watson algorithm on a point set $A \subset \mathbb{R}^2$ produces a Delaunay triangulation of A.*

Proof. Correctness of the Bowyer Watson algorithm on a point set A follows almost directly from Theorem 2.4. We begin the algorithm by creating a super triangle, which is certainly the Delaunay triangulation of 3 points - the vertices of the super triangle. Since the super triangle surrounds A, we will always insert new points into the interior of the current triangulation. Proceeding inductively, by Theorem 2.4, we will have a Delaunay triangulation after each point insertion. If A contains n points, then before we remove the super triangle from the triangulation, we will have the Delaunay triangulation of $n+3$ points. It is easy to observe that the Delaunay triangulation of $A \cup B$ gives the Delaunay triangulation of A when all triangles with vertices in B are removed. Thus, removing the all triangles from the triangulation which share vertices with the super triangle leaves the Delaunay triangulation of A. □

3 Voronoi Diagram Construction

3.1 Two Dimensions: Pseudocode

The following is basic pseudo code for constructing the Voronoi diagram given a Delaunay triangulation in two dimensions.

```
//Data: Delaunay triangulation of point set A - del, a set of Polygons - voro.
//Result: voro will be the Voronoi diagram induced by A.
Polygon_set voronoi(Triangle_set del) {
  Polygon_set voro;
  for(each triangle T in del) {
    find the circumcentre of T, and store in T;
  }
  for(each point x in A) {
    compute the voronoi region of x and store in voro;
  }
  return voro;
}
```

3.2 Obtaining a Voronoi Region

The method for computing the voronoi region of a particular point $x \in A$ is to start at a Delaunay triangle with x as a vertex - then move in a particular direction along a strongly connected set of Delaunay triangles that contain x and adding the Voronoi vertices encountered as vertices of the Voronoi region of x. One could think of this as 'turning' around the vertex, see Figure 8a. If the Voronoi region of x is bounded, we will return to the starting Delaunay triangle, and stop there. If the Voronoi region is unbounded, we will have to travel as far as we can in one direction from the starting triangle then stop and return to the starting triangle, proceeding to travel in the other direction. This will pick up two additional vertices, one for each Delaunay edge containing x that lies on $H(A)$ which make the polygon unbounded. These additional vertices are points on the perpendicular bisector of an edge of $H(A)$. A true Voronoi vertex will always be the circumcentre of a Delaunay triangle. See Figure 8b for two example Voronoi regions.

3.3 Unbounded Voronoi Edges

Consider constructing an unbounded Voronoi edge from a Delaunay edge e in a triangle T, which is labelled anti-clockwise, or positively oriented. Let the e be directed to agree with the orientation of T. There are three cases:

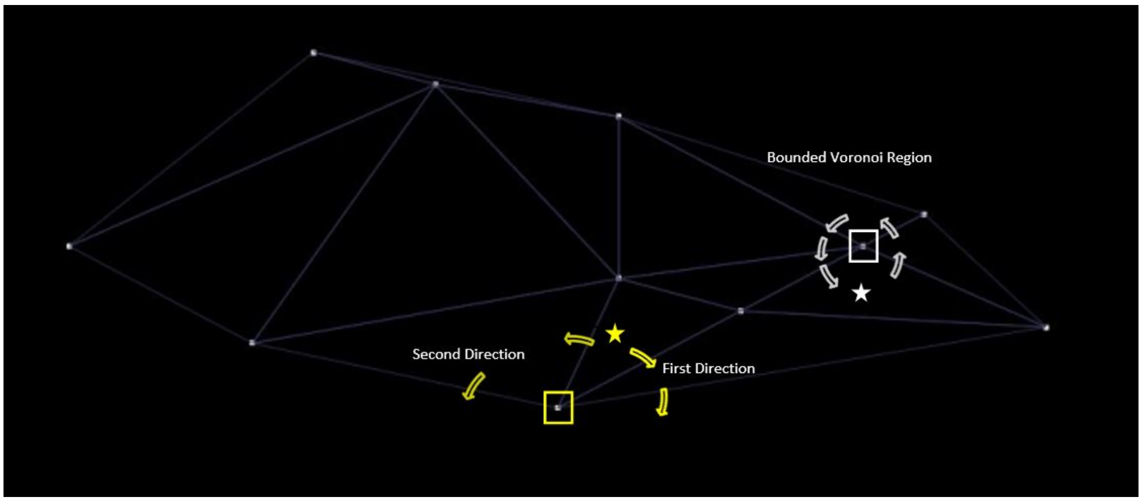
1. The circumcentre of T lies on the side of e so that e and the circumcentre of T form an anti-clockwise triangle. See Figure 9a.
2. The circumcentre of T lies on the side of e so that e and the circumcentre of T form a clockwise triangle. See 9a.
3. The circumcentre of T lies on e. See Figure 9b.

In the first case, the Voronoi edge will be a ray starting at the circumcentre of T going in the direction of the vector from the circumcentre of T to the midpoint of e. In the second case the Voronoi edge will again be a ray starting at the circumcentre of T but the direction of the ray is reversed from the first case. In the final case, the midpoint of e and the circumcentre of T are the same. Thus the perpendicular bisector of e must be calculated, and a ray drawn along the perpendicular bisector starting at the circumcentre of T and directed away from T itself.

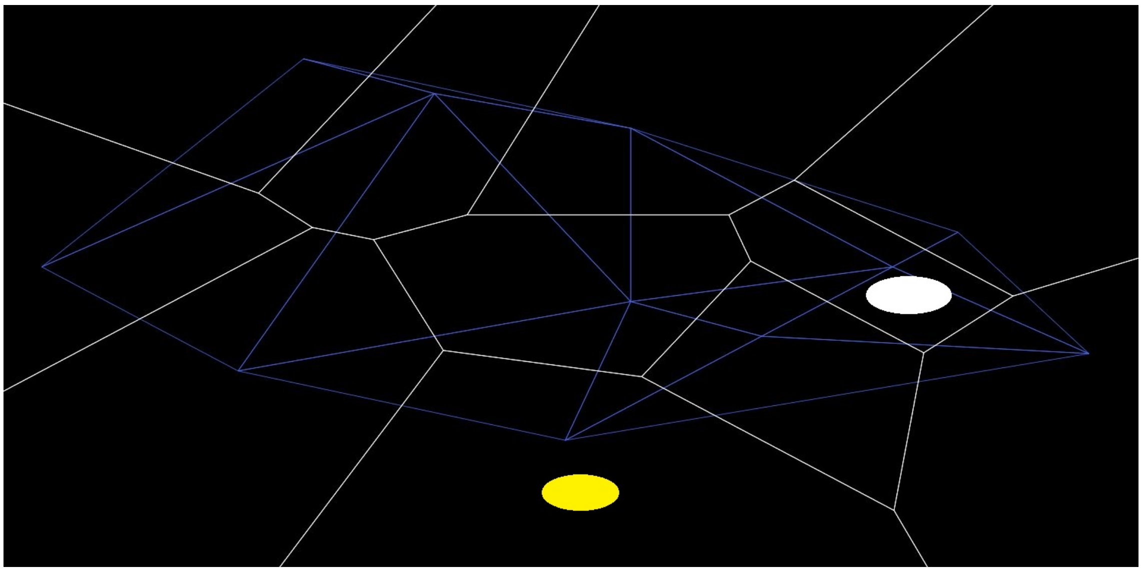
Figure 9 demonstrates this construction, the unbounded Voronoi edges are shown in blue. In Figure 9a 1, 2 and 3 are the vertices are of a Delaunay triangle, 4 is the circumcentre of that triangle and 123 is an anti-clockwise triangle. For the edge 12, 124 forms a clockwise triangle, so the Voronoi edge corresponding to edge 12 is formed according to case 2. However, for the edges 23 and 31, 234 and 314 form anti-clockwise triangles so the Voronoi edges corresponding to 23 and 31 are formed according to case 1. In Figure 9b, the circumcentre of the triangle lies on an edge e of the triangle, so the Voronoi edge corresponding to e is formed according to case 3.

3.4 Three Dimension Specifics

The Voronoi diagram construction in three dimensions is very similar to two dimensions. We form a Voronoi polygon in three dimensions in a very similar manner to

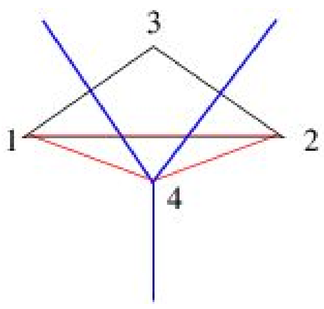


(a) A Delaunay triangulation showing the paths (white and yellow arrows) taken to construct the Voronoi region of two Delaunay vertices. The stars indicate the starting triangles for those points.

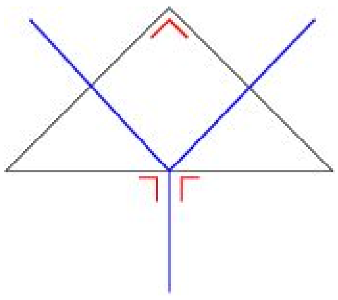


(b) The Voronoi diagram corresponding to the above Delaunay triangulation. Notice the Voronoi regions for the two points above are constructed from Voronoi vertices encountered along the paths. The regions are marked by yellow and white ovals.

Figure 8: Demonstrating constructing a bounded and an unbouded Voronoi region



(a) Cases 1 and 2



(b) Case 3

Figure 9: The different cases for constructing a Voronoi edge from a Delaunay edge.

how a Voronoi region was formed in two dimensions. Instead of turning about a Delaunay vertex in 2D we turn about a Delaunay edge in 3D, forming a Voronoi polygon in the same manner - picking up the Voronoi vertices corresponding to the tetrahedrons we meet, with two additional vertices for unbounded polygons. These polygons are, very importantly, guaranteed to be co-planar and convex as is pointed out in [Led07]. A Voronoi region in 3D for a point v is the polyhedron formed by all the Voronoi polygons corresponding to Delaunay edges which have v as a vertex. The following is short pseudocode to demonstrate this.

```
//Data: Delaunay tetrahedrilisation of point set A - del, a set of Polyhedrons -
    voro.
//Data: an Edge_set es and a Polygon p.
//Result: voro will be the Voronoi diagram induced by A.
Polyhedron_set voronoi(Tetrahedron_set Del) {
  Polyhedron_set voro; Edge_set es; Polygon p;
  for(each tetrahedron T in del) {
    find the circumcentre of T, and store in T;
  }
  for(each point x in A) {
    find all edges in del which contain x and store in es;
    pick the Polyhedron in voro corresponding to x, call it ph;
    for(each edge e in es) {
      compute Voronoi polygon corresponding to e, storing in P;
      add P as a face of ph;
  }
  return voro;
}
```

4 Important Formulae

4.1 Orientation Check

A triangle is positively oriented if walking along the boundary of the triangle in the direction of the orientation keeps the interior of the triangle on your left, and negatively oriented otherwise. A positively oriented triangle has a positive signed area in 2D, likewise a negatively oriented triangle has negative signed area. Orientation will be very important to us for two reasons:

1. Finding the orientation of the triangle with vertices ABC allows us to determine which side of the line AB that C lies on.

2. We will lift the vertices of a triangle to a parabaloid to check if a triangle satisfies the empty circle property. This test relies on the triangle being positively oriented.

We begin by computing the signed area of the triangle ABC up to a positive scale factor.

$$\text{SignedArea}(A,B,C) = \begin{vmatrix} A_x & A_y & 1 \\ B_x & B_y & 1 \\ C_x & C_y & 1 \end{vmatrix}$$

And define our orientation test by:

$$\text{PositiveOriented}(A,B,C) = \begin{cases} 1, & \text{if SignedArea}(A,B,C) > 0 \\ 0, & \text{if SignedArea}(A,B,C) < 0 \\ -\infty, & \text{if SignedArea}(A,B,C) = 0 \end{cases}$$

Remark. In the third case of PositiveOriented's definition we do not have a triangle, but in fact have a line. Hence the value $-\infty$. If we ever try to create a triangle ABC and get a value of -∞ for PostiveOriented(A, B, C), then the program will throw an error.

When dealing with a tetrahedron $ABCD$ we instead compute the signed volume of the tetrahedron up to a positive scale factor.

$$\text{SignedVolume}(A, B, C, D) = \begin{vmatrix} A_x & A_y & A_z & 1 \\ B_x & B_y & B_z & 1 \\ C_x & C_y & C_z & 1 \\ D_x & D_y & D_z & 1 \end{vmatrix}$$

Translating a tetrahedron does not change its signed volume, and so translation by $-D$ reduces the above to:

$$\begin{vmatrix} A_x - D_x & A_y - D_y & A_z - D_z & 1 \\ B_x - D_x & B_y - D_y & B_z - D_z & 1 \\ C_x - D_x & C_y - D_y & C_z - D_z & 1 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} A_x - D_x & A_y - D_y & A_z - D_z \\ B_x - D_x & B_y - D_y & B_z - D_z \\ C_x - D_x & C_y - D_y & C_z - D_z \end{vmatrix}$$

The orientation test for a tetrahedron is the same as for a triangle with SignedArea(A, B, C) replaced by SignedVolume(A, B, C, D).

4.2 Empty Circle Check

We need to efficiently check if a point D lies inside the circumcircle of a triangle T. To do this, we will end up finding the signed volume of a parallelepiped. The main result is from Guibas and Stolfi [GS85] (see pg. 107), Figure 10 is extracted from this. The idea is to lift D and the vertices of T to the parabaloid of revolution by the following map, and perform an orientation test on the tetrahedron defined by the four points.

$$\begin{aligned} \mathbb{R}^2 &\to \mathbb{R}^3 \\ (a, b) &\mapsto (a, b, a^2 + b^2) \end{aligned}$$

Lemma 4.1. *The point D lies inside of the circumcircle of a positively oriented triangle ABC if and only if*

$$\mathscr{D}(A, B, C, D) = \begin{vmatrix} A_x & A_y & A_x^2 + A_y^2 & 1 \\ B_x & B_y & B_x^2 + B_y^2 & 1 \\ C_x & C_y & C_x^2 + C_y^2 & 1 \\ D_x & D_y & D_x^2 + D_y^2 & 1 \end{vmatrix} > 0$$

We will not transcribe the proof of the above lemma here, but a picture (Figure 10) from [GS85] gives some intuition. We define a plane P in three dimensions using the vertices of the triangle lifted onto the parabaloid of revolution. Another point x is co-circular with the vertices of the triangle in 2D if and only if it is co-planar with P when lifted onto the parabaloid of revolution. If x is not co-planar, then the side of the plane x lies on determines if x lies inside the circle.

Translating a triangle ABC and a point D in the plane will not change the orientation of ABC or the sign of $\mathscr{D}(A, B, C, D)$, and so we can reduce our test to

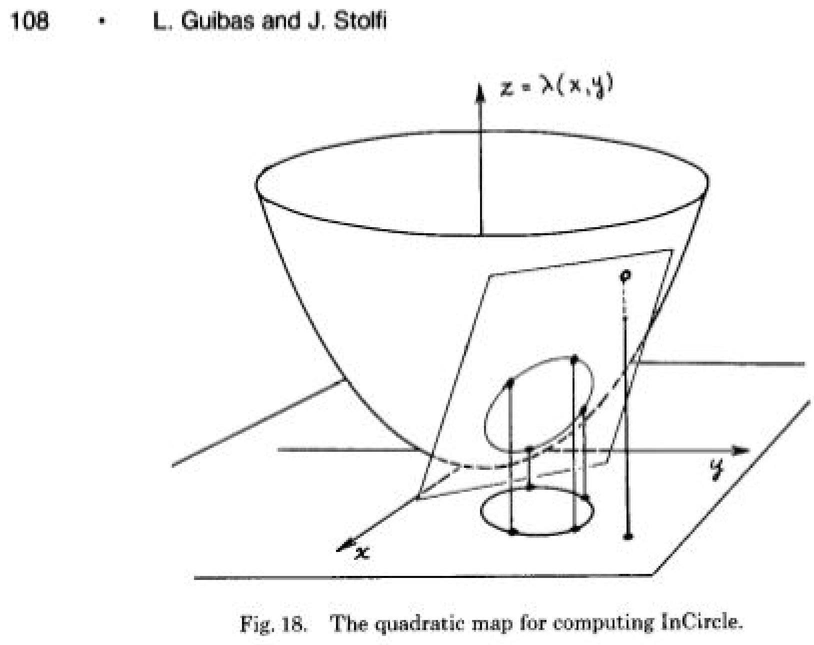

108 • L. Guibas and J. Stolfi

Fig. 18. The quadratic map for computing InCircle.

Figure 10: Figure of the lifting from Guibas and Stolfi [GS85].

a 3×3 determinant. We translate by $-D$, to obtain that the point D lies inside of the circumcircle of a positively oriented triangle ABC if and only if

$$\begin{vmatrix} A_x - D_x & A_y - D_y & (A_x - D_x)^2 + (A_y - D_y)^2 & 1 \\ B_x - D_x & B_y - D_y & (B_x - D_x)^2 + (B_y - D_y)^2 & 1 \\ C_x - D_x & C_y - D_y & (C_x - D_x)^2 + (C_y - D_y)^2 & 1 \\ 0 & 0 & 0 & 1 \end{vmatrix} > 0$$

Which is, expanding on the last row

$$\begin{vmatrix} A_x - D_x & A_y - D_y & (A_x - D_x)^2 + (A_y - D_y)^2 \\ B_x - D_x & B_y - D_y & (B_x - D_x)^2 + (B_y - D_y)^2 \\ C_x - D_x & C_y - D_y & (C_x - D_x)^2 + (C_y - D_y)^2 \end{vmatrix} > 0$$

So we end up checking if the signed volume of the parallelepiped defined by the three 3D vectors that are the rows of the above matrix is postive. The above generalises to three dimensions by mapping (a, b, c) to $(a, b, c, a^2 + b^2 + c^2)$.

Lemma 4.2. *The point E lies inside the circumcircle of a positively oriented tetrahedron $ABCD$ if and only if*

$$\begin{vmatrix} A_x - E_x & A_y - E_y & A_z - E_z & (A_x - E_x)^2 + (A_y - E_y)^2 + (A_z - E_z)^2 \\ B_x - E_x & B_y - E_y & B_z - E_z & (B_x - E_x)^2 + (B_y - E_y)^2 + (B_z - E_z)^2 \\ C_x - E_x & C_y - E_y & C_z - E_z & (C_x - E_x)^2 + (C_y - E_y)^2 + (C_z - E_z)^2 \\ D_x - E_x & D_y - E_y & D_z - E_z & (D_x - E_x)^2 + (D_y - E_y)^2 + (D_z - E_z)^2 \end{vmatrix} > 0$$

4.3 Circumcentre Calculation

Finding the circumcentre of a triangle sitting in $\mathbb{R}^2$ is relatively easy. One can hardcode finding the intersection of the perpendicular bisectors of two sides of the triangle, giving the circumcentre. However for a triangle sitting in $\mathbb{R}^3$, or a tetrahedron, some more sophistication is needed. The following are from Shewchuk [She13]:

Let $|A|$ denote the Euclidean norm of the vector A. Let $A \times B$ denote the cross product of A and B. The circumcentre of the circumcircle of a triangle ABC in $\mathbb{R}^3$ is coplanar with the triangle and is given by:

$$C + \frac{[|A-C|^2(B-C) - |B-C|^2(A-C)] \times [(A-C) \times (B-C)]}{2|(A-C) \times (B-C)|^2}$$

The circumcentre of the circumsphere of a tetrahedron $ABCD$ in $\mathbb{R}^3$ is given by:

$$D + \frac{|A-D|^2(B-D) \times (C-D) + |B-D|^2(C-D) \times (A-D) + |C-D|^2(A-D) \times (B-D)}{2\text{SignedVolume}(A,B,C,D)}$$

Where

$$\text{SignedVolume}(A,B,C,D) = \begin{vmatrix} A_x - D_x & A_y - D_y & A_z - D_z \\ B_x - D_x & B_y - D_y & B_z - D_z \\ C_x - D_x & C_y - D_y & C_z - D_z \end{vmatrix}$$

5 Data Structures and Efficiency

5.1 Data Structures

Some main data structures used in the program will be outlined here as an aid. Edges, Faces, Triangles and Tetrahedron structures will have vertices as integer labels rather than actual real number co-ordinates as it is easier to compare two integers than multiple doubles and retrieval is also easier. The integer labels can pull from an array of doubles when co-ordinates are necessary. We consider a triangle to have the following data structure.

```c
typedef struct Edge {
  int from, to; //2 vertices as integer labels
} Edge;

typedef struct Triangle {
  int vert_1, vert_2, vert_3; //Three vertices of triangle as integer labels
  Edge edge_1, edge_2, edge_3; //Three edges of the triangle
  //adjacent[i] is pointer to the triangle sharing edge_i+1 - NULL if none exist
  struct Triangle* adjacent[3];
  double circum[2]; //Circumcentre of the triangle
  int checked; //Has the triangle been checked in step j of Bowyer Watson?
} Triangle;
```

We use a doubly linked list (DLL) to hold pointers to the triangles in Del, and our bad triangles. Insertion and deletion are quite fast, and being able to search in both directions is very powerful when Hilbert sorting is introduced.

```c
typedef struct DLL_NODE {
  Triangle *data;
  struct DLL_NODE *next, *prev;
} DLL_NODE;

struct DLL{
  DLL_NODE *first, *last;
};
```

We use an edge stack to form our polygon of edges on the boundary of the Delaunay cavity. While a triangle pointer stack keeps track of the good triangles that sit on the border of the Delaunay cavity. A stack is used for these because an edge on the boundary only needs to be seen once and is then thrown away.

```c
//Edge stack
typedef struct Stack{
  int top_index;
  int capacity;
  Edge *item;
} Stack;
//Triangle pointer stack
typedef struct Triangle_Stack{
  int top_index;
  int capacity;
  Triangle **item;
} Triangle_Stack;
```

We give a Voronoi region in 2D the following data structure. The Voronoi Diagram will be an array of pointers to Voronoi regions, or Polygons.

```c
struct Polygon2D { //An array of vertices, and a boolean unbounded
  double* v;
  int capacity;
  int num_items;
  int unbounded;
};
```

Much is similar in the three dimensional case, with a DLL still holding the Delaunay tetrahedrilisation, and a polygon stack and tetrahedron pointer stack holding the boundary of the Delaunay cavity and the tetrahedrons on that boundary respectively. However the Voronoi region structure is different, and we need a tetrahedron data structure, not a triangle data structure. Let us start with the tetrahedron data structure. To quote [LS05], which compared five Delaunay tetrahedrilisation programs, "All five programs store the set of tetrahedra, and for each tetrahedron t, references to its vertices and neighbors—a neighbor is another tetrahedron that shares a common triangle with t", and we are no different.

```c
typedef struct Edge { //Edges have integer labelled vertices
  int v[2]; //2 vertices as integer labels
} Edge;

typedef struct Face { //Faces are triangles - integer labelled vertices
  int v[3]; //3 vertices as integer labels
} Face;

typedef struct Tetrahedron {
  int v[4]; //Four vertices of tetrahedron as integer labels.
  Face f[4]; //Four faces of tetrahedron
  //adj[i] is pointer to the Tetrahedron sharing face[i] - NULL if none exist.
  struct Tetrahedron* adj[4];
  double circum[3]; //The co-ordinates of the circumcentre
  int checked; //Has the tetrahedron has been checked in step j of bowyer_watson
} Tetrahedron;
```

A Voronoi region in 3D will be a polyhedron. Again, the Voronoi diagram will be an array of pointers to Voronoi regions, in this case Polyhedron structures.

```c
typedef struct Polygon3D {//Holds the Polygon's vertices in an array.
  double* v;
  int capacity;
  int num_items;
} Polygon3D;

struct Polyhedron {//An array of Polygon3D pointers, and a boolean unbounded
  Polygon3D **faces;
  int num_faces;
  int unbounded;
};
```

5.2 Keeping Track of Adjacencies

The basic Bowyer Watson implementation works, but it is rather slow. The first problem we have is that for each point in A we loop over every triangle in Del. Using the fact that Delaunay cavities must be strongly connected (see Lemma 2.2) we see that looping over every triangle in Del to find all bad triangles is unnecessary. We can instead find just one element of the cavity, and then recursively check the neighbours of this triangle to find the entire cavity.

The following keeps track of triangle adjacencies on the fly, the first two operations are performed at each step of the Boywer Watson, with the last perfomed only when removing the super triangle.

1. Keep track of which good triangles sit on the border of the Delaunay cavity, call them the border triangles. Then, when a new triangle T is formed with an edge e the border triangle BT sharing e (if it exists) points to T, and T points back to BT.

2. When all the new triangles have been added to retriangulate the cavity, we fill in the adjacencies between these new triangles.

3. On removal of all triangles which share vertices with the super triangle, the triangles which were adjacent to removed triangles must have this adjacency set to NULL.

5.3 Hilbert Curve

From the previous section we see that it is imperative that we quickly locate one bad triangle, as the other bad triangles spread out as neighbours from the first one. This brings us on to the notion of point location. If we can sort the input points such that the points are entered into the Bowyer Watson algorithm with the point entered at step n geometrically close to the point entered at step $n+1$ this will have two benefits. Firstly, we will likely locate a bad triangle quickly since the newly added point will probably be in the circumcenter of a recently added triangle to the Delaunay triangulation. Secondly, these recently added triangles in question can be cached and accessed faster. I was reading lecture notes from Remacle and Legat at Université catholique de Louvain [RL15] when I encountered the idea of sorting the input points along a Hilbert curve.

Space filling curves are commonly used to reduce a multi-dimensional problem to a one dimensional problem, producing a mapping from a hypercube to an interval. A curve is a linear traversal of a discrete multi-dimensional space. A Hilbert curve is a continuous, space-filling curve - that is a curve with no breaks or jumps whose range fills a hypercube. The Hilbert curve can be thought of as the limit of a sequence of curves $(H_n)_{n=1}^{\infty}$, where n denotes the order of the curve. To get next curve in the sequence we take the curve of order $n-1$ and make four copies of it. We then rotate and place these copies so that one copy sits in each quadrant of a square, with the bottom left quadrant's curve starting at $(0,0)$ and the bottom right quadrant's curve finishing at $(n^2-1,0)$. We join up these curves in the order: bottom left, top left, top right, bottom right to form the curve of order n. The curves of order 1 and 2 are depicted in Figures 11 and 12 in two dimensions, with H_1 being the first curve in the sequence $(H_n)_{n=1}^{\infty}$.

Each of the curves H_n is simple, meaning H_n does not cross itself. However the Hilbert curve, the limit of the sequence of curves $(H_n)_{n=1}^{\infty}$, is not simple. If it were simple, then there would be a continuous bijection from the unit interval to the unit square. The unit interval is a compact space by the Heine-Borel theorem since it is closed and bounded. The unit square is a Hausdorff space since it is a subset of the Hausdorff space $\mathbb{R}^2$. So this bijection would in fact be a homeomorphism, as we will show. However, the unit interval can't be homeomorphic to the unit square as the unit interval can be made disconnected by removing an interior point from the unit interval, but the unit square can not be made disconnected by removing a single point from it.

Theorem 5.1. *A continuous bijection from a compact space C to a Hausdorff space H is a homeomorphism.*

Proof. Let f be a continuous bijection from C to H. Let $V \subset C$ be closed in C. Then V is compact since V is a closed subset of a compact space. Thus $f(V)$ is compact since the image of a compact space under a continuous map is compact. Finally $f(V)$ is closed in H since a compact subset of a Hausdorff space is in fact closed. But $f(V) = (f^{-1})^{-1}(V)$, that is, the image of V under f is the preimage of V under the inverse of f. Thus if V is closed in C, $(f^{-1})^{-1}(V)$ is closed in H, so f^{-1} is continous. So f is a homeomorphism. $\square$

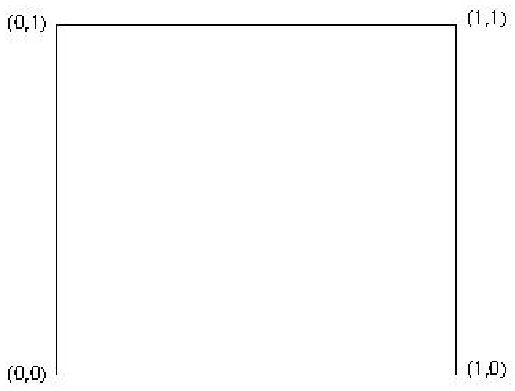


Figure 11: Order 1 curve

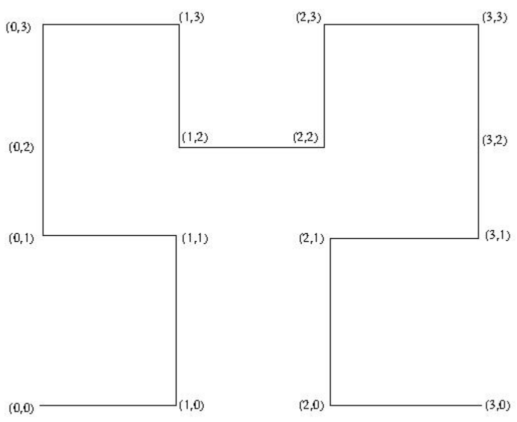


Figure 12: Order 2 curve

We choose the Hilbert curve in particular because it has very nice properties, such as being space-filling as was already mentioned. Most importantly, two points that are close along a Hilbert curve in two dimensions are guaranteed to be close together in the plane. Namely we are interested in encoding each point in $A \subset \mathbb{R}^2$ to a one dimensional distance, and sorting A in increasing order of these distances. There are many good algorithms for encoding and decoding Hilbert curves, with eight presented in [Liu+16] alone. We implement the two dimensional Hilbert Curve encoding algorithm from [CWS07].

```c
//rotate/flip a quadrant appropriately.
void rot(int quad, int *x, int *y, int w) {
  int temp;
  if (quad == 0) {
    temp = *x;
    *x = *y;
    *y = temp;
  }
  else if (quad == 1) {
    *y = *y - w;
  }
  else if (quad == 2) {
    *x = *x - w;
    *y = *y - w;
  }
  else {
    temp = *x;
    *x = w - *y - 1;
    *y = w * 2 - temp - 1;
  }
}

//Converts a 2D integer point (x,y) to a 1D distance d, with grid resolution n
//d will be called the Hilbert distance
int xy2d (int n, int x, int y) {
  int r;
  int max;
  int w;
  int temp;
  int quad;
  int rx, ry, d = 0;
  if (x >= y) max = x;
  else max = y;
  r = floor(log(max)/log(2)) + 1;
  w = (int)pow(2, r - 1);
  if ((n % 2) != (r % 2)) {
    temp = x;
    x = y;
    y = temp;
  }
  while (r != 0) {
    rx = (x & w) > 0;
    ry = (y & w) > 0;
    quad = (3 * rx) ^ ry;
    d += w * w * quad;
    rot(quad, &x, &y, w);
    r = r - 1;
    w = w/2;
  }
  return d;
}
```

To explain this, n is the maximum order curve we will consider and r is the minimum order curve such that (x, y) sits on the curve. Initially, if the parity of n and r differ we swap x and y. Then using bitwise operations, we find which quadrant of the square $(0, 0)$, $(0, r^2 - 1)$, $(r^2 - 1, r^2 - 1)$, $(r^2 - 1, 0)$ that (x, y) sits in. Based on this quadrant, the encoded value d of (x, y) is updated, and (x, y) is updated in the function *rot* to account for the rotation of the curves in each different quadrant. Quadrant 0 is the bottom left, quadrant 1 is the top left, quadrant 2 is the top right, quadrant 3 is the bottom right. Then r is decremented because we need to check a curve one order lower than we had in the last step. We iteratively repeat this procedure until r is 0.

To Hilbert sort our points in $A \subset \mathbb{R}^2$ we can make a copy of A and perform a translation on all the points (x, y) so that $x \geq 0$ and $y \geq 0$, then convert these to integers by forming an integer point (a, b) with, say the first five significant figures of x and y respectively. Sorting these integer points in ascending ordering of their Hilbert distance, will thus sort A. This will not be an exact sorting of A due to an inaccurate integer conversion, but that is acceptable as two adjacent points in A after sorting will still be close geometrically, which is the end goal. See Figure 13 for an example of this sorting, the white line shows the path taken to traverse the points of A after sorting.

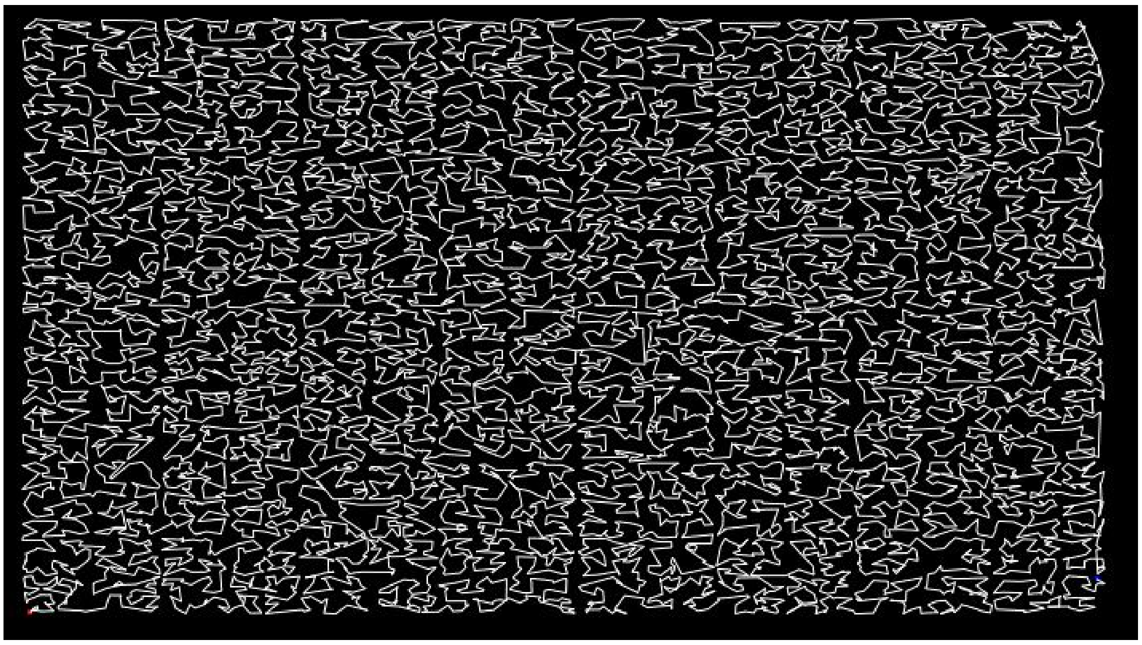


Figure 13: 10,000 points in 2D sorted along a Hilbert curve

5.4 Time savings

To demonstrate the magnitude of the time savings, here is an example on 20,000 points in $\mathbb{R}^2$. The following experiments were performed on my home computer with an Intel Core i5-6500 CPU and 8GB of RAM. Without performing the first optimisation, that is finding one bad triangle and finding the other bad triangles as neighbours of it, the program took 51.4 seconds. Performing the first optimisation, improved this to 17.4 seconds. Performing a Hilbert sort on the 20,000 points before using the optmised Bowyer Watson algorithm vastly improved the program run time to 0.6 seconds altogether for the sorting, Delaunay triangulation and Voronoi diagram computation.

This allows the program to run on very large data sets very fast. There is one problem, the program does not use exact arithmetic, and so it often makes critical errors on data sets with more than about 100,000 points - this is generally due to a triangle approximating a line. In Figure 14 there is a zoomed in section of a Voronoi diagram of 100,000 points. The full Voronoi diagram of the points took 3.1 seconds to compute.

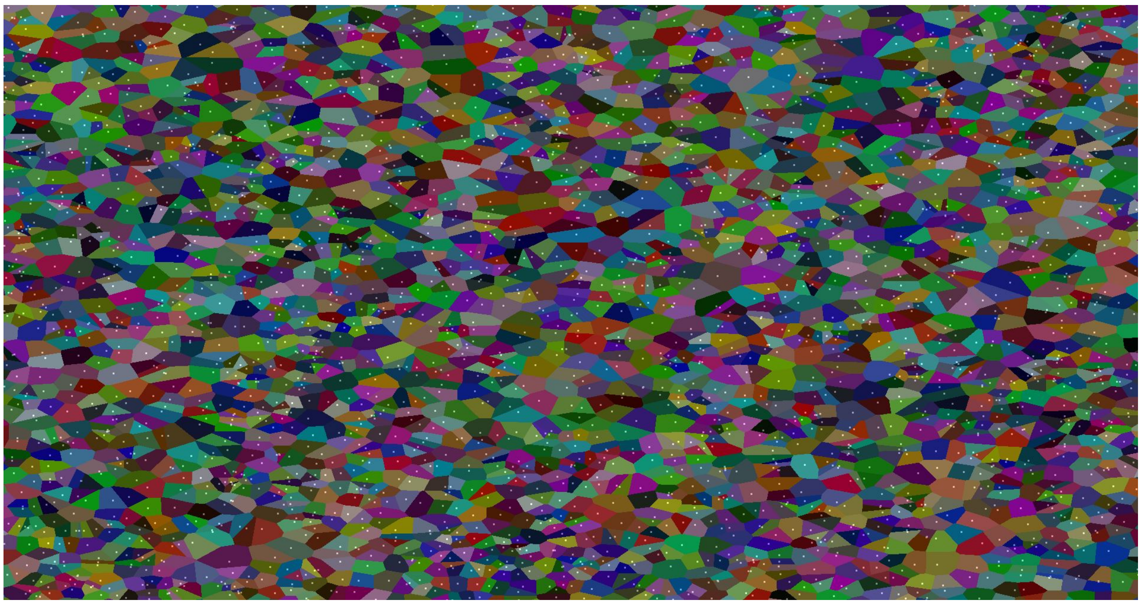


Figure 14: A section cut out a Voronoi diagram of 100,000 points in 2D

6 Results and Applications

6.1 Results in Two Dimensions

An example of the program output on ten points in two dimensions follows:

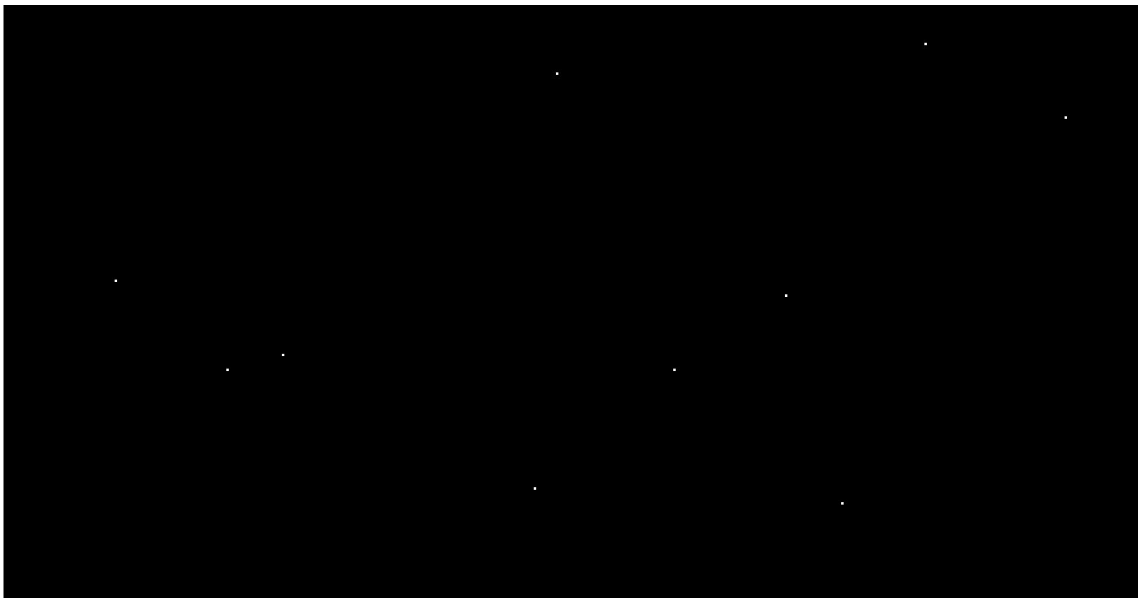


Figure 15: 10 input points in 2D

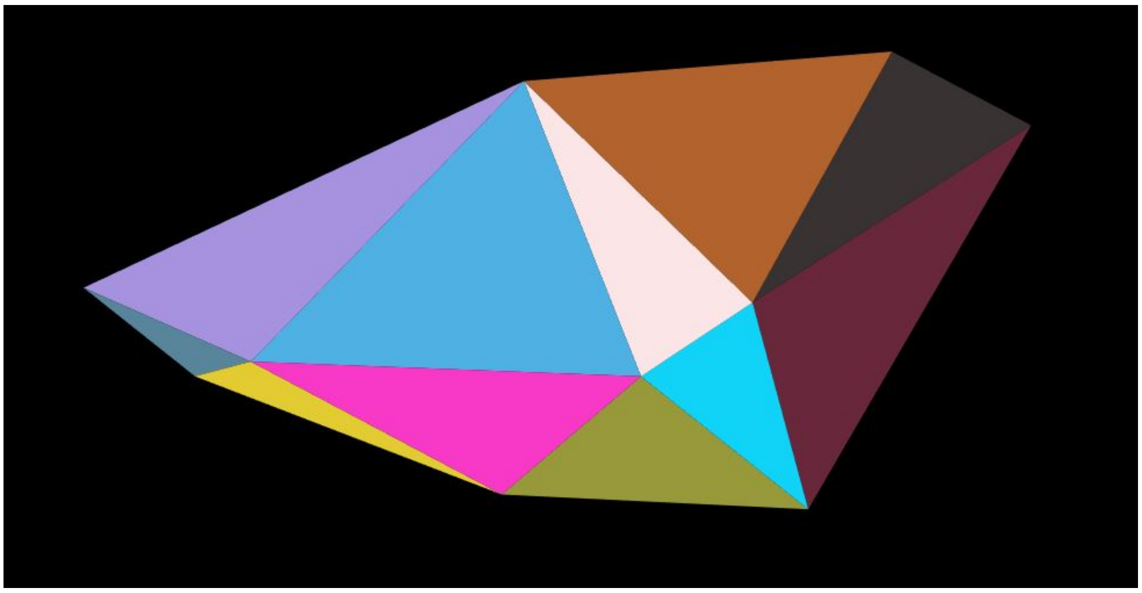


Figure 16: The Delaunay triangulation of the 10 points

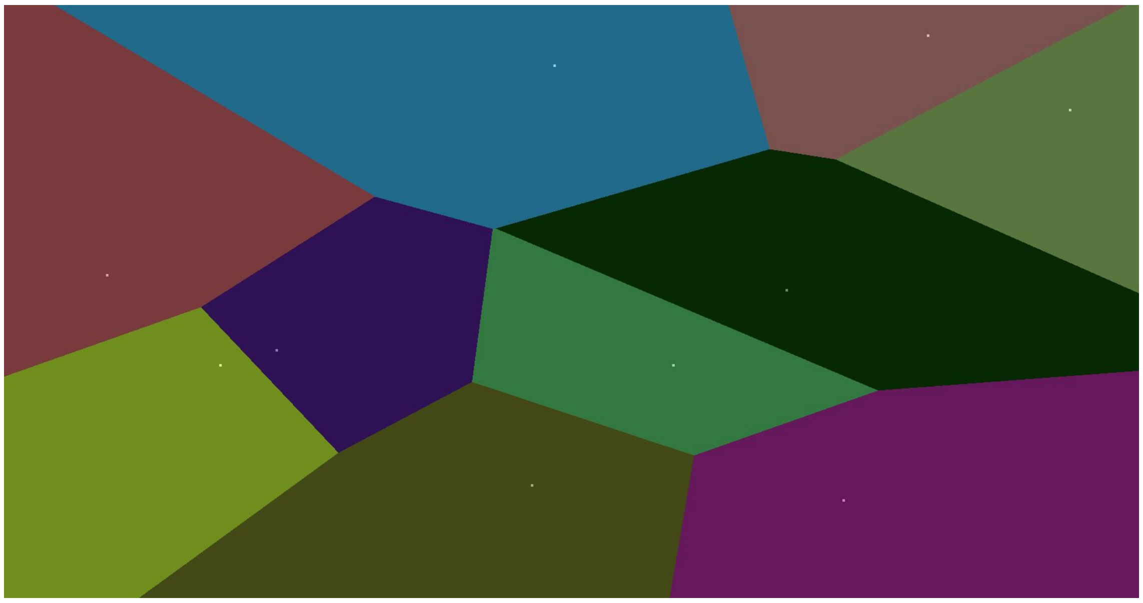


Figure 17: The Voronoi diagram of the 10 points

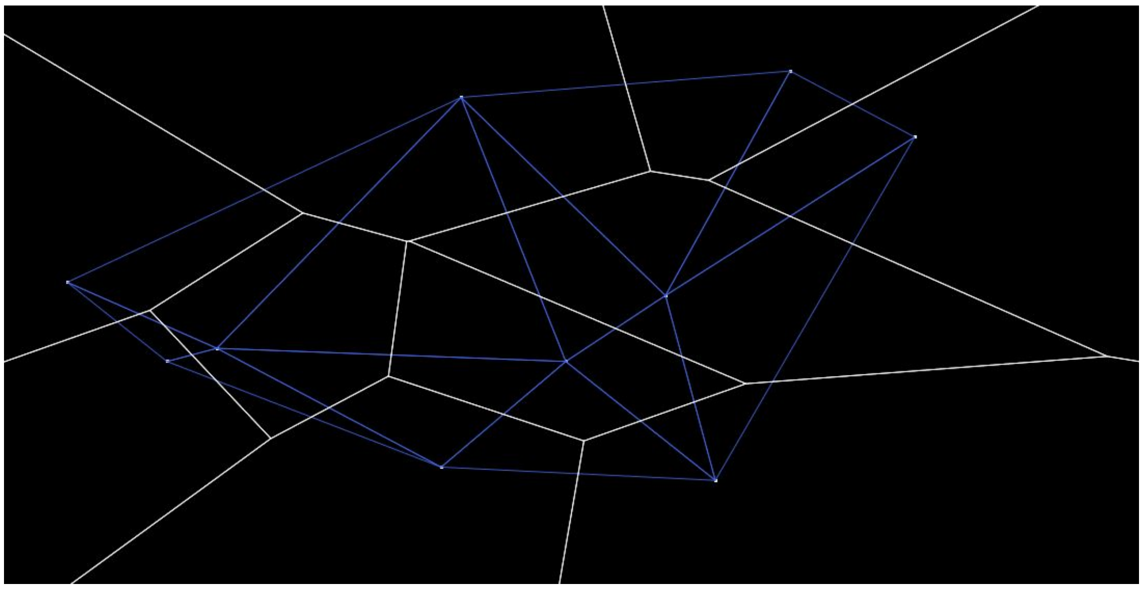


Figure 18: The Voronoi diagram of the points in white, with the Delaunay in blue

An example of the program output on 100 points in two dimensions follows:

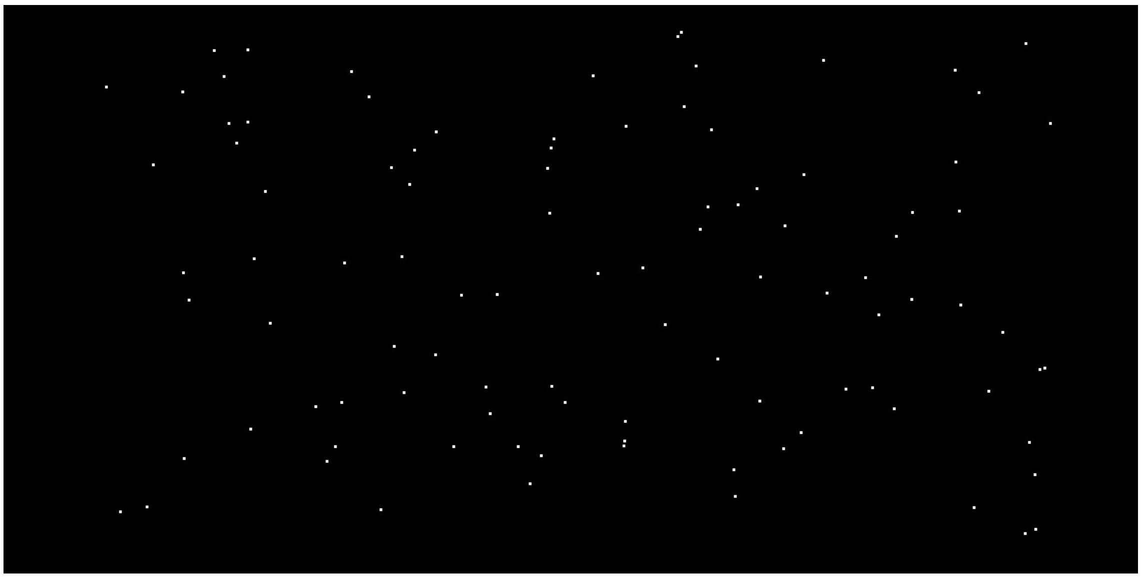


Figure 19: 100 input points in 2D

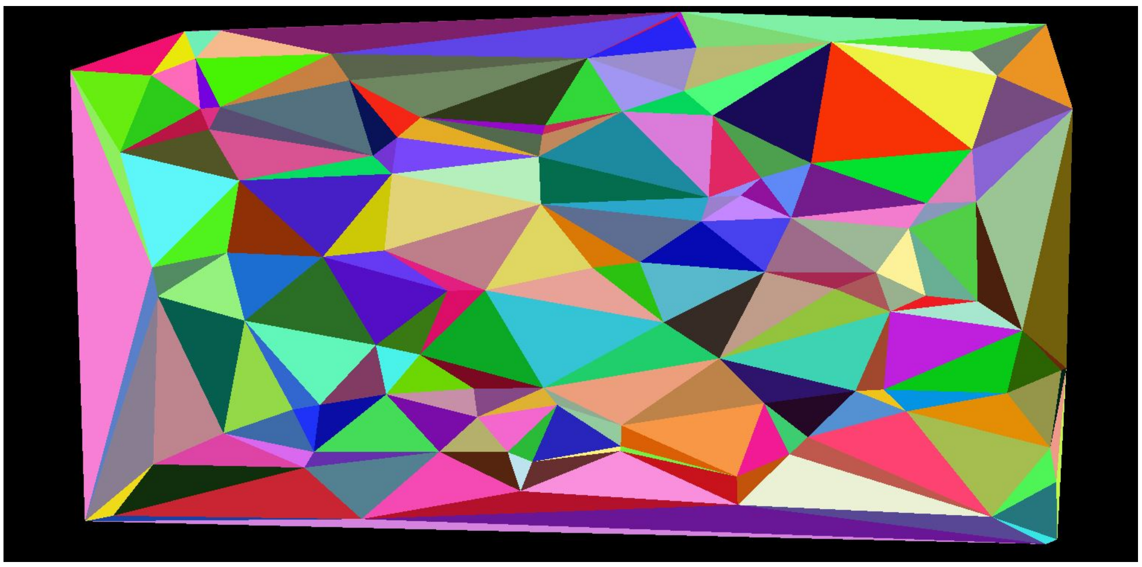


Figure 20: The Delaunay triangulation of the 100 points

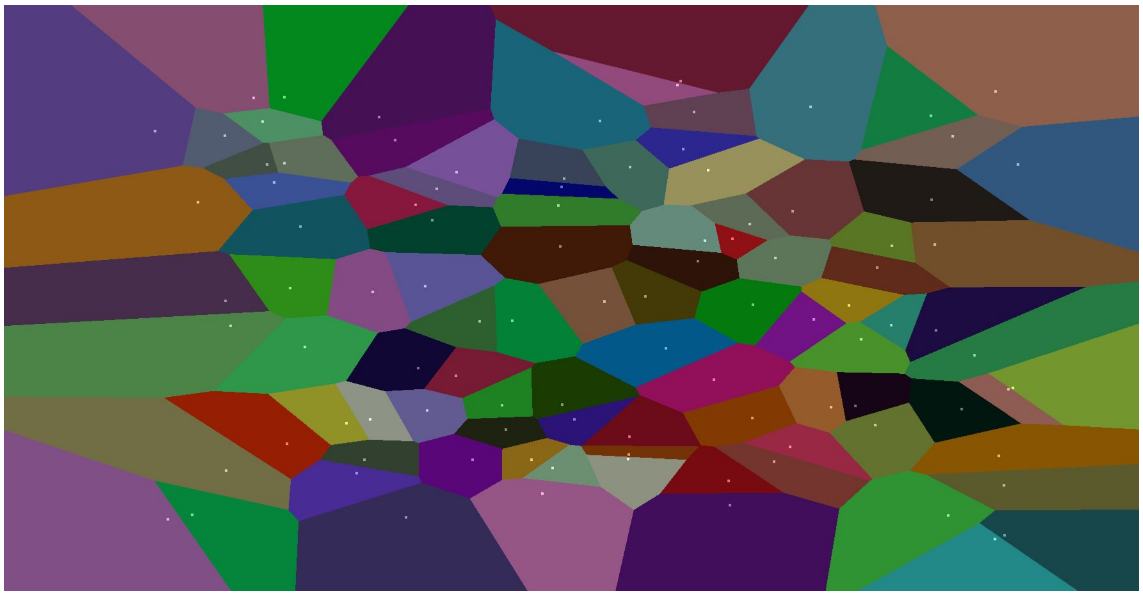


Figure 21: The Voronoi diagram of the 100 points

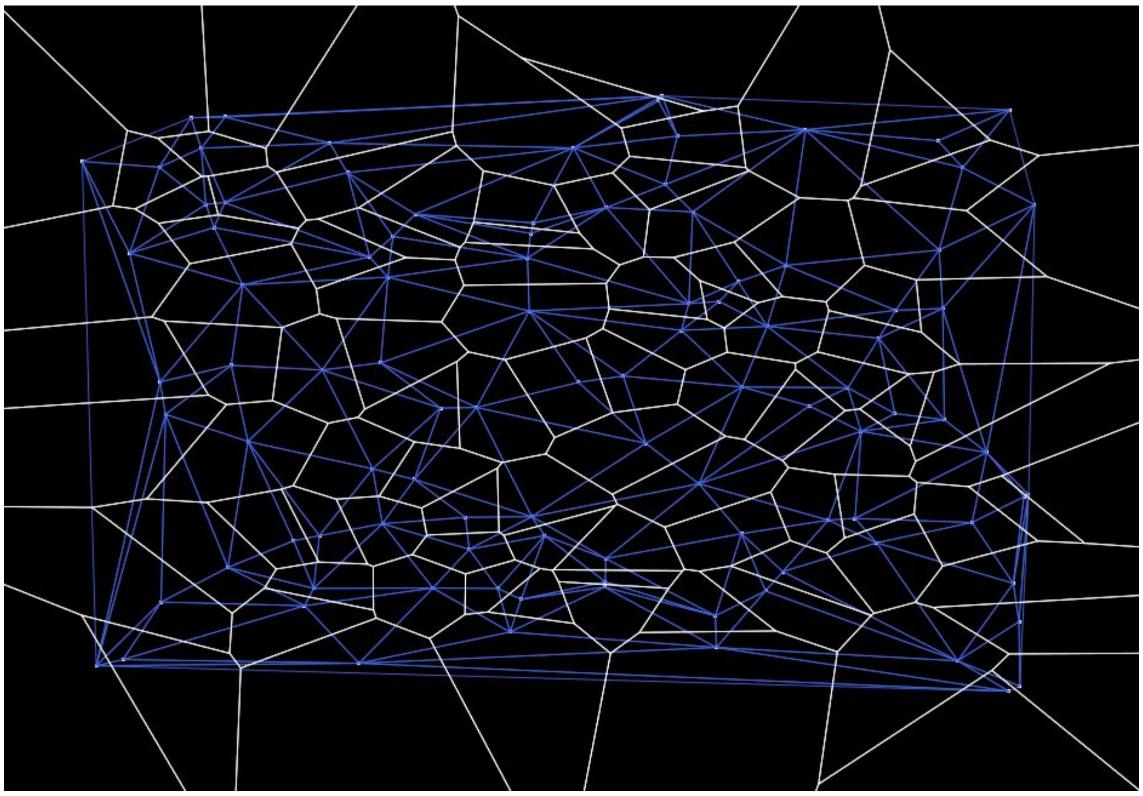
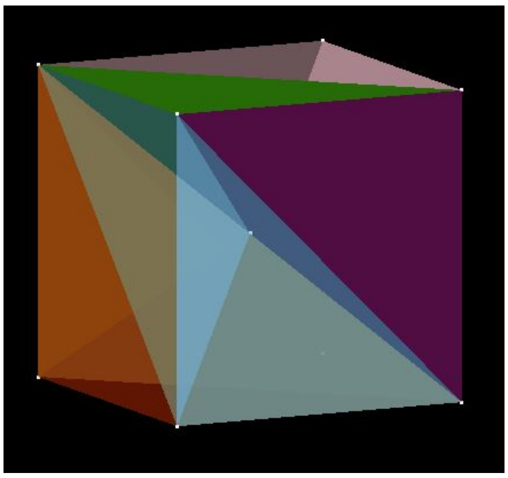


Figure 22: The Voronoi diagram of the points in white, with the Delaunay in blue.

6.2 Results in Three Dimensions

An example of the program output on nine points consiting of the eight vertices of the unit cube, and the centre of the unit cube which is the origin in $\mathbb{R}^3$ follows.


(a) Colour version

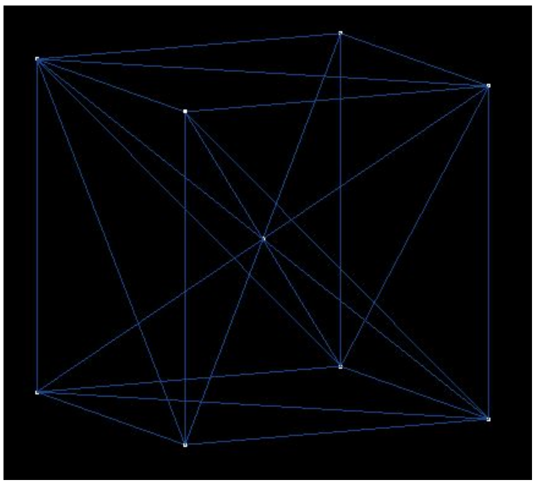
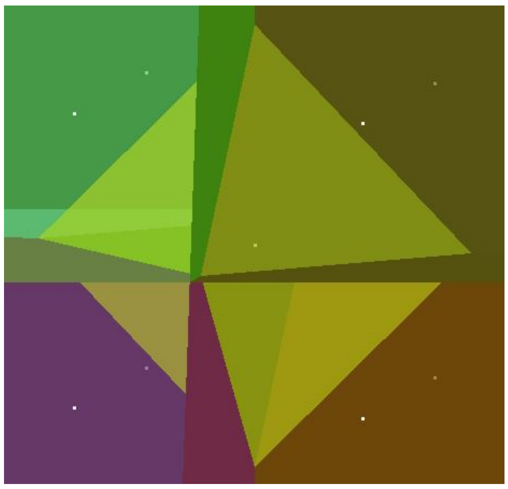

(b) A mesh version

Figure 23: The Delaunay tetrahedrilisation of the nine points, containing 12 tetrahedrons - two for each face of the cube.


(a) The Voronoi diagram of the nine points

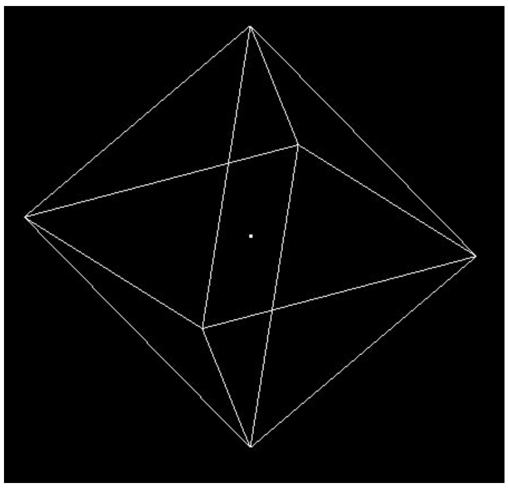
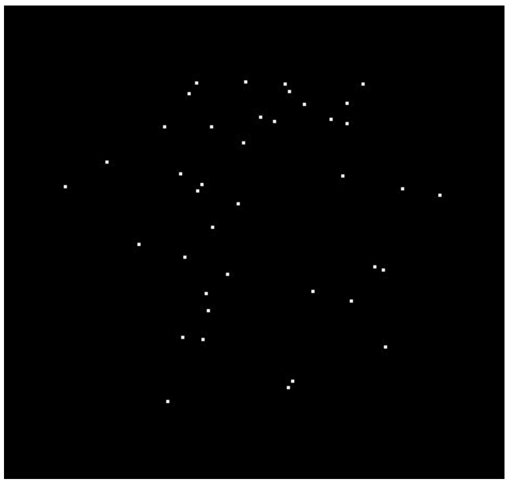
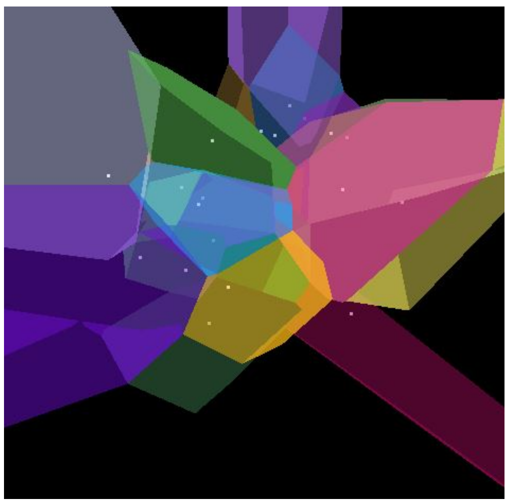

(b) The only bounded Voronoi region

Figure 24: The bounded region shown in Figure 24b corresponds to the centre of the cube.


(a) Forty random points in 3D


(b) Bounded Voronoi regions of these points

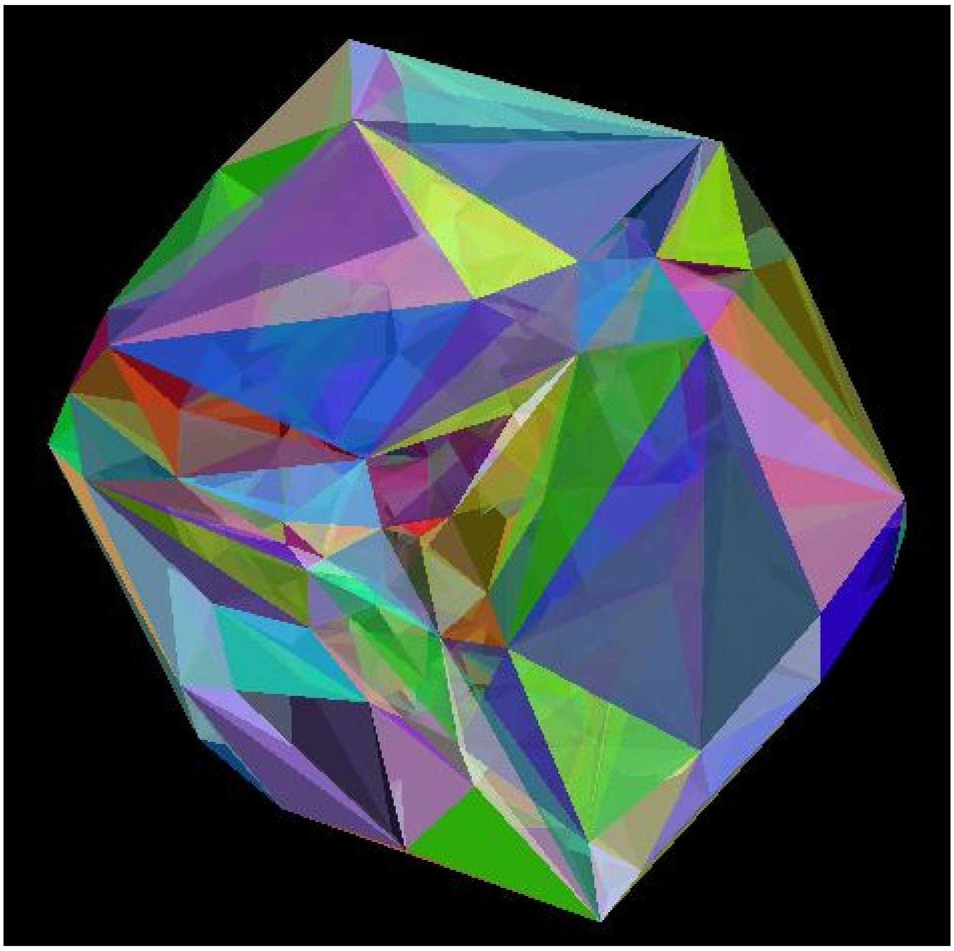


Figure 26: The Delaunay tetrahedrilisation of 1000 points

6.3 Applications

We briefly discuss some applications of Voronoi Diagrams:

1. Texture generation: The Voronoi Diagram induced by point sets can be used to produce natural textures in computer graphics such as lava like textures, or cobblestone flooring.

2. Natural growth models: Voronoi diagrams can be considered as arising from the following process. A set of points each begin growing a crystal. The crystals move outwards from the points at the same rate. Crystals stop growing in a particular direction if they are touching another crystal in that direction. This will result in each crystal being a Voronoi region. See [LP12], Section 3 for examples in biology modelled with Voronoi diagrams.

3. Geostatistics: For example, say you took measurements of the amount of gold at one hundred exploratory drill points in a region of mountains. Forming the Voronoi diagram associated with these one hundred points would give a method of estimating the area in these mountains with the highest concentration of gold deposits.

7 Problems and Conclusion

7.1 Implementation Flaws

We do not use exact arithmetic in our implementation, because it is slow. However for a robust implementation exact arithmetic would be a necessity. Not using it causes two main problems:

1. We assume that input points are distinct, but we do not check this assumption is valid. Even if we did check this, without exact arithmetic we would have to say two points that are within a certain tolerance are non-distinct.

2. The program can currently fail to correctly compute an orientation. In two dimensions this primarily occurs when a triangle is very thin or when a tetrahedron on the parabaloid of revolution is very thin. As a result the program can incorrectly determine if a point lies inside the circumcircle of a triangle, or draw a voronoi edge in the wrong direction. There is an informative discussion of this problem in [RL15] (see pg. 26).

Currently we do not sort the points along a Hilbert curve in three dimensions. This is because I found visually presenting the three dimensional Voronoi diagram to be difficult, and from what I found, so do most. The program already runs fast enough without sorting for any input point set I can reasonably visualise. However, if I improved the visualisation it would definitely be worth sorting the points.

On the topic of sorting, it is suggested in the literature to avoid completely sorting the input points but instead to sort randomised bins of points. This is an effort to combine the benefits of randomised point insertion, and inserting points in an order so that they are close together. See [ACR03] for more information on this.

7.2 Problems Encountered

This section is a bit different from what would be regularly seen in articles and perhaps somewhat informal, but I would like to outline some things I found particularly hard to work with in this project and how that effected the project.

1. It is hard to pick good data structures, yet very important to do so. As an example, when I first wrote the program in two dimensions the triangle data structure did not store which triangles were adjacent to it. This turned out to be very inconvenient later when efficiency was considered, and took much time to fix. Furthermore, the Delaunay triangulation was originally stored in a bag, and this had to be changed to a doubly linked list when ordering became important.

2. Getting the Voronoi diagram from the Delaunay triangulation was actually harder than I expected it to be. It is very easy to determine the Voronoi edge between neighbouring triangles, but I originally found it quite awkward when a Delaunay triangle had an edge with no neighbour. The problem was figuring out which side of an edge the circumcentre of the triangle lay on. Fortunately, performing an orientation check on the triangle formed by the edge and the circumcentre easily solves this.

3. Learning OpenGL was somewhat problematic. I am very grateful to my supervisor for his C code for drawing three dimensional convex hulls with OpenGL, but there are still some flaws in my visualisation. For example, in three dimensions, where two tetrahedrons or Voronoi regions intersect, the colours of the region merge on the face they intersect on, which is not ideal. There are some clever ways of getting a face to be different colours on different sides of the face, but I did not get it quite right.

7.3 Conclusion

We have discussed in detail how to compute the Delaunay triangulation and the Voronoi diagram of a point set in $\mathbb{R}^2$ and shown how this adapts to $\mathbb{R}^3$. We have shown how to improve the efficiency of the Bowyer Watson algorithm using the fact that the triangles in a Delaunay cavity are strongly connected, and using Hilbert sorting to order the input points. Finally we have presented results, and applications of the work. Full C source code is provided in the appendix for the three dimensional case, with code snippets for the two dimensional. At my website, www.maths.tcd.ie/~martins7 I will host a project folder as long as I have access to the URL, and full two dimensional source code can be found there.

To conclude, computational geometry is not easy, but it is as rewarding as it is challenging. I would like to quote the Wikipidea page on Computational Geometry here, "Some [geometry] problems seem so simple that they were not regarded as problems at all until the advent of computers". That is to say, some geometry problems seem awfully simple until you try to get a computer, which lacks our powerful visual processing, to solve them.

Bibliography

[Des44] R. Descartes. *Principia philosophiae.* Ludovicum Elzevirium, 1644.

[Bow81] A. Bowyer. "Computing Dirichlet tessellations". In: *The Computer Journal* 24.2 (1981), pp. 162–166.

[Wat81] D. F. Watson. "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes". In: *The Computer Journal* 24.2 (1981), pp. 167–172.

[GS85] Leonidas Guibas and Jorge Stolfi. "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams". In: *ACM Transactions on Graphics* 4.2 (1985), pp. 74–123.

[Far95] Andrew Farrell. *Fortune's Voronoi Diagram Algorithm Exteded to Convex Sites.* Trinity College, 1995.

[AK00] Franz Aurenhammer and Rolf Klein. "Voronoi Diagrams". In: *Handbook of Computational Geometry* 5 (2000), pp. 201–290.

[Mat02] Jiří Matoušek. *Lectures on discrete geometry.* Vol. 212. Springer Science & Business Media, 2002.

[ACR03] Nina Amenta, Sunghee Choi, and Günter Rote. "Incremental constructions con BRIO". In: *Proceedings of the nineteenth annual symposium on Computational geometry.* ACM. 2003, pp. 211–219.

[LS05] Yuanxin Liu and Jack Snoeyink. “A Comparison of Five Implementations of 3D Delaunay Tesselation”. In: *Combinatiorial and Computational Geometry* 52 (2005), pp. 439–458.

[CWS07] Ningtao Chen, Nengchao Wang, and Baochang Shi. “A new algorithm for encoding and decoding the Hilbert order”. In: *Software: Practice and Experience* 37.8 (2007), pp. 897–908.

[HÓDY07] Paul Harrington, Colm Ó Dúnlaing, and Chee K Yap. “Optimal Voronoi diagram construction with n convex sites in three dimensions”. In: *International Journal of Computational Geometry & Applications* 17.06 (2007), pp. 555–593.

[Led07] Hugo Ledoux. “Computing the 3d Voronoi diagram robustly: An easy explanation”. In: *Voronoi Diagrams in Science and Engineering, 2007. ISVD'07. 4th International Symposium on.* IEEE. 2007, pp. 117–129.

[LRS10] Jesús A. De Loera, Jörg Rambau, and Francisco Santos. *Triangulations Structures for Algorithms and Applications.* Springer, 2010.

[DO11] Satyan L. Devadoss and Joseph O'Rourke. *Discrete and Computational Geometry.* Princeton University Press, 2011.

[LP12] Thomas M. Liebling and Lionel Pournin. “Voronoi Diagrams and Delaunay Triangulations: Ubiquitois Siamese Twins”. In: *Documenta Mathematica* (2012), pp. 419–431.

[Buc+13] Kevin Buchin et al. “Vertex deletion for 3D Delaunay triangulations”. In: *European Symposium on Algorithms.* Springer. 2013, pp. 253–264.

[She13] Jonathan R. Shewchuk. *Lecture notes on Geometric Robustness.* `https://people.eecs.berkeley.edu/~jrs/meshpapers/robnotes.pdf`. [Online; accessed 06-March-2017]. 2013.

[RL15] Jean-Francois Remacle and Vincent Legat. *Construction of 2D Delaunay Triangulations.* `http://perso.uclouvain.be/jean-francois.remacle/LMECA2170/Chap2.pdf`. [Online; accessed 16-January-2017]. 2015.

[Liu+16] Hui Liu et al. “Encoding and Decoding Algorithms for Arbitrary Dimensional Hilbert Order”. In: *arXiv preprint arXiv:1601.01274* (2016).

Appendix: Code

Short Header File

```c
#ifndef HEADER_FILE
#define HEADER_FILE

typedef struct DLL DLL;
typedef struct Polyhedron Polyhedron;
typedef struct Polygon_2D Polygon_2D;

#endif
```

Three Dimensional Code

```c
//Windows compile gcc 3D_Voronoi.c -o 3D_Voronoi glut32.lib -lopengl32 -lglu32
//Unix compile gcc 3D_Voronoi.c -o 3D_Voronoi -lglut -lGL -lGLU -lm
//Sean Martin TCD with Colm O Dunlaing- 18/03/2017
//Contact info: martins7@tcd.ie or seankieran.m@hotmail.com

#ifdef _WIN32
#define _CRT_SECURE_NO_DEPRECATE
#include <windows.h>
#endif

#include <time.h>
#include <stdio.h>
#include <GL/gl.h>
#include <GL/glu.h>
#ifdef _WIN32
#include <C:\Users\Sean\Google Drive\College Mathematics\SS\Voronoi Diagrams\glut.h>
#else
#include <GL/glut.h>
#include <unistd.h>
#endif
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <float.h>
#include "Voronoi.h"

//Global variables
double *p; //p for points - our array of points
double *color; //holds the colors of the Delaunay tetrahedron
int state = 0; //A kbd function use
int tumble_on = 0; //A kbd function use
int num_points; //Will hold the number of points in consideration
static int debug = 0; //on/off - debugging/ not debugging
int p_on = 0, d_on = 0, v_on = 0, cull_on = 0; //Drawing states
double tumble = 0, r_x = 0, r_y = 0, zoom = 1; //for animation
double bmin[3], bmax[3]; //for clipping
DLL *del; //Stores Delaunay tetrahedron
Polyhedron **voro; //Stores Voronoi Diagram

typedef struct Polygon_3D {//Holds the Polygons vertices in an array.
  double* v;
  int capacity;
  int num_items;
} Polygon_3D;

struct Polyhedron {//An array of Polygon_3D pointers, and a boolean unbounded
  Polygon_3D **faces;
  int num_faces;
  int unbounded;
};

//Makes a polygon with capacity cap
Polygon_3D *make_polygon(int cap) {
  Polygon_3D *pg = malloc(sizeof(Polygon_3D));
  if (pg == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  pg->capacity = cap;
  pg->v = malloc(sizeof(double) * pg->capacity);
  if (pg->v == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  pg->num_items = 0;
  return pg;
}

//makes a polyhedron with n faces;
Polyhedron *make_polyhedron(int n) {
  int i;
  Polyhedron *ph = malloc(sizeof(Polyhedron));
  ph->num_faces = n;
  if (ph == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  ph->faces = malloc(sizeof(Polygon_3D*) * ph->num_faces);
  if (ph->faces == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  for ( i = 0; i < ph->num_faces; ++i) {
    ph->faces[i] = make_polygon(30);
  }
  ph->unbounded = 0;
  return ph;
}

//adds the point a to the polygon pg
void add_to_polygon(double *a, Polygon_3D *pg) {
  if((pg->num_items + 3) > pg->capacity) {
    pg->capacity *= 2;
    pg->v = realloc(pg->v, sizeof(double) * pg->capacity);
  }
  pg->v[pg->num_items++] = a[0];
  pg->v[pg->num_items++] = a[1];
  pg->v[pg->num_items++] = a[2];
}

//pn pulls co-ordinate n from the point labelled by integer input.
double p0(int input) {
  return p[3 * input];
}
double p1(int input) {
  return p[3 * input + 1];
}
double p2(int input) {
  return p[3 * input + 2];
}

//Calculates determinant of 2x2 matrix with rows [a1, a2], [b1, b2]
double determinant_2(double a1, double a2, double b1, double b2) {
  return (a1 * b2) - (a2 * b1);
}

//Calculates the determinant of the 3x3 matrix with rows a,b,c
double determinant_3(double *a, double *b, double *c) {
  return a[0] * (b[1] * c[2] - b[2] * c[1]) +
         a[1] * (b[2] * c[0] - b[0] * c[2]) +
         a[2] * (b[0] * c[1] - b[1] * c[0]);
}

//Calculates the determinant of the 4x4 matrix with rows a, b, c, d
double determinant_4(double *a, double *b, double *c, double *d) {
  double result = 0;
  double f[3], g[3], h[3];
  f[0] = b[1]; f[1] = b[2]; f[2] = b[3];
  g[0] = c[1]; g[1] = c[2]; g[2] = c[3];
  h[0] = d[1]; h[1] = d[2]; h[2] = d[3];
  result += a[0] * determinant_3(f, g, h);
  f[0] = b[0]; f[1] = b[2]; f[2] = b[3];
  g[0] = c[0]; g[1] = c[2]; g[2] = c[3];
  h[0] = d[0]; h[1] = d[2]; h[2] = d[3];
  result -= a[1] * determinant_3(f, g, h);
  f[0] = b[0]; f[1] = b[1]; f[2] = b[3];
  g[0] = c[0]; g[1] = c[1]; g[2] = c[3];
  h[0] = d[0]; h[1] = d[1]; h[2] = d[3];
  result += a[2] * determinant_3(f, g, h);
  f[0] = b[0]; f[1] = b[1]; f[2] = b[2];
  g[0] = c[0]; g[1] = c[1]; g[2] = c[2];
  h[0] = d[0]; h[1] = d[1]; h[2] = d[2];
  result -= a[3] * determinant_3(f, g, h);
  return result;
}

/*****************************************************************************
These data structures have vertices as integer labels, which will pull
from an array of points when actual co-ordinates are needed.
*****************************************************************************/
typedef struct Edge { //Edges have integer labelled vertices
  int v[2]; //2 vertices as integer labels
} Edge;

typedef struct Face { //Faces are triangles - integer labelled vertices
  int v[3]; //3 vertices as integer labels
} Face;

typedef struct Tetrahedron {
  int v[4]; //Four vertices of tetrahedron as integer labels.
  Face f[4]; //Four faces of tetrahedron
  //adj[i] is pointer to the Tetrahedron sharing face[i] - NULL if none exist.
  struct Tetrahedron* adj[4];
  double circum[3]; //The co-ordinates of the circumcentre
  int checked; //Has the tetrahedron has been checked in step j of bowyer_watson
} Tetrahedron;

//Calculates the orientation determinant of the Tetrahedron
//It is reduced to a 3x3 calculation through translation by -t->v[3]
double tetrahedron_determinant(Tetrahedron *t) {
  double a[3], b[3], c[3];
  a[0] = p0(t->v[0]) - p0(t->v[3]);
  a[1] = p1(t->v[0]) - p1(t->v[3]);
  a[2] = p2(t->v[0]) - p2(t->v[3]);
  b[0] = p0(t->v[1]) - p0(t->v[3]);
  b[1] = p1(t->v[1]) - p1(t->v[3]);
  b[2] = p2(t->v[1]) - p2(t->v[3]);
  c[0] = p0(t->v[2]) - p0(t->v[3]);
  c[1] = p1(t->v[2]) - p1(t->v[3]);
  c[2] = p2(t->v[2]) - p2(t->v[3]);
  return determinant_3(a, b, c);
}

//makes a positively oriented Tetrahedron with vertices v0, v1, v2, v3.
Tetrahedron *make_tetrahedron(int v0, int v1, int v2, int v3) {
  Tetrahedron *t = malloc(sizeof(Tetrahedron));
  if (t == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  t->v[0] = v0; t->v[1] = v1; t->v[2] = v2; t->v[3] = v3;
  double check = tetrahedron_determinant(t);
  //check should be greater than 0 is the tetrahedron is positively oriented.
  //reversing labels reverses the sign of check - and thus the orientation
  if(check < 0) {
    t->v[0] = v1;
    t->v[1] = v0;
  }
  else if(check == 0) {
    fprintf(stderr, "ERROR: Tried to make coplanar tetrahedron\n");
    exit(-1);
  }
  //Storing the faces - each face is oriented correctly
  t->f[0].v[0] = t->v[0];
  t->f[0].v[1] = t->v[1];
  t->f[0].v[2] = t->v[2];
  t->f[1].v[0] = t->v[2];
  t->f[1].v[1] = t->v[1];
  t->f[1].v[2] = t->v[3];
  t->f[2].v[0] = t->v[2];
  t->f[2].v[1] = t->v[3];
  t->f[2].v[2] = t->v[0];
  t->f[3].v[0] = t->v[0];
  t->f[3].v[1] = t->v[3];
  t->f[3].v[2] = t->v[1];

  t->adj[0] = t->adj[1] = t->adj[2] = t->adj[3] = NULL;
  t->circum[0] = t->circum[1] = t->circum[2] = 0;
  t->checked = -1;
  return t;
}

/*****************************************************************************
In the bowyer_watson algorithm we start by using a big tetrahedron which
surrounds all of the points in consideration.
This big tetrahedron must be removed at the end of the algorithm, so this
function checks if Tetrahedron *t shares a vertex with this big tetrahedron -
the big tetrahedron has vertices n, n + 1 and n + 2
*****************************************************************************/
int shares_vertex_supert(Tetrahedron *t, int n) {
  int b = t->v[0];
  if ((b == n) || (b == n + 1) || (b == n + 2) || (b == n + 3))
    return 1;

  b = t->v[1];
  if ((b == n) || (b == n + 1) || (b == n + 2) || (b == n + 3))
    return 1;

  b = t->v[2];
  if ((b == n) || (b == n + 1) || (b == n + 2) || (b == n + 3))
    return 1;

  b = t->v[3];
  if ((b == n) || (b == n + 1) || (b == n + 2) || (b == n + 3))
    return 1;

  return 0;
}

//Checks if faces f1 == f2, that is have the same vertices
int equal(Face f1, Face f2) {
  if ((f1.v[0] + f1.v[1] + f1.v[2]) != (f2.v[0] + f2.v[1] + f2.v[2])) {
    return 0;
  }
  int a;
  a = f1.v[0];
  if ((a != f2.v[0]) && (a != f2.v[1]) && (a != f2.v[2])) {
    return 0;
  }
  a = f1.v[1];
  if ((a != f2.v[0]) && (a != f2.v[1]) && (a != f2.v[2])) {
    return 0;
  }
  a = f1.v[2];
  if ((a != f2.v[0]) && (a != f2.v[1]) && (a != f2.v[2])) {
    return 0;
  }
  return 1;
}

//Check is edges e1 == e2, that is have the same vertices.
int equal_edges(Edge e1, Edge e2) {
  if((e1.v[0] != e2.v[0]) && (e1.v[0] != e2.v[1])) return 0;
  if((e1.v[1] != e2.v[0]) && (e1.v[1] != e2.v[1])) return 0;
  return 1;
}

```

```
//Code for a DLL of Tetrahedron pointers follows
typedef struct DLL_NODE {
  Tetrahedron *data;
  struct DLL_NODE *next, *prev;
} DLL_NODE;

struct DLL{
  DLL_NODE *first, *last;
};

DLL_NODE *make_node() {
  DLL_NODE *new = malloc(sizeof(DLL_NODE));
  if (new == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  new->next = NULL;
  new->prev = NULL;
  new->data = NULL;
  return new;
}

DLL *make_list() {
  DLL *new = malloc(sizeof(DLL));
  if (new == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  new->first = NULL;
  new->last = NULL;
  return new;
}

int is_empty_list(DLL *list) {
  return (list->first == NULL);
}

void add_to_list (Tetrahedron *t, DLL *list) {
  DLL_NODE *node = make_node();
  node->data = t;
  if (is_empty_list(list)) {
    list->last = node;
  }
  else {
    list->first->prev = node;
  }
  node->next = list->first;
  list->first = node;
}

void remove_node (DLL_NODE *node, DLL *list) {
  if (is_empty_list(list)) {
    fprintf(stderr,"ERROR tried to delete from empty list\n");
    exit(-1);
  }
  if (node == (list->first)) {
    list->first = node->next;
  }
  else {
    node->prev->next = node->next;
  }
  if (node == (list->last)) {
    list->last = node->prev;
  }
  else {
    node->next->prev = node->prev;
  }
  free(node);
}

DLL_NODE *find_node (Tetrahedron *t, DLL *list) {
  DLL_NODE *node;
  node = list->first;
  while(node != NULL) {
    if (node->data == t) {
      return node;
    }
    node = node->next;
  }
  fprintf(stderr,"Tetrahedron not present in list %p", t);
  exit(-1);
}

int list_length (DLL *list) {
  int count = 0;
  DLL_NODE *node;
  node = list->first;
  while(node != NULL) {
    ++count;
    node = node->next;
  }
  return count;
}

void empty_dll (DLL *list) {
  DLL_NODE *node;
  DLL_NODE *temp;
  node = list->first;
  while (node != NULL) {
```

```c
    temp = node;
    node = node->next;
    free(temp->data);
    free(temp);
  }
  list->first = NULL;
  list->last = NULL;
}

//Print the contents of the doubly linked list, *list.
void print_DLL(DLL *list) {
  int n;
  int k;
  DLL_NODE *node;
  FILE *file;

  if(list == del) {
    if ((file = fopen("Logs/Graph_3D.txt","w")) == NULL) {
      fprintf(stderr, "File not openable \n");
      exit(-1);
    }
  }
  else {
    if ((file = fopen("Logs/DLL_3D.txt","w")) == NULL) {
      fprintf(stderr, "File not openable \n");
      exit(-1);
    }
  }
  fprintf(file, "num tetrahdrons: %d\n", list_length(list));
  node = list->first;
  while (node != NULL) {
    fprintf(file, "Tetrahedron vertices:\n");
    for (k = 0; k < 4; ++k) {
      n = node->data->v[k];
      fprintf(file, "%d -- %lf, %lf, %lf\n", n, p0(n), p1(n), p2(n));
    }
    node = node->next;
  }
  if(fclose(file) == EOF) {
    fprintf(stderr, "Couldn't close file");
    exit(-1);
  }
}

//The code for a push down Face stack follows
typedef struct Stack{
  int top_index;
  int capacity;
  Face *item;
} Stack;

Stack *make_stack() {
  Stack *s  = (Stack*) malloc(sizeof(Stack));
  if (s == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  s->top_index = -1;
  s->capacity = 100;
  s->item = (Face*) malloc(s->capacity * sizeof(Face));
  if (s->item == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  return s;
}

int is_empty_stack (Stack *s) {
  return ( s->top_index == -1 );
}

Face top (Stack *s) {
  if(!is_empty_stack(s)) {
    return s->item [s->top_index];
  }
  else {
    fprintf(stderr,"ERROR top called on empty stack");
    exit(-1);
  }
}

void push (Face t, Stack *s) {
  int i = s -> top_index + 1;
  if ( i >= s -> capacity ) {
    s->capacity *= 2;
    s->item = realloc(s->item, s->capacity * sizeof(Face));
  }
  s -> item[i] = t;
  s -> top_index = i;
}

void pop (Stack *s) {
  -- (s -> top_index);
}

void free_stack(Stack *s) {
  free(s->item);
  free(s);
}
```

```c

//Code for a pushdown Tetrahedron pointer stack follows
typedef struct T_Stack{
  int top_index;
  int capacity;
  Tetrahedron **item;
} T_Stack;

T_Stack *make_t_stack() {
  T_Stack *s  = (T_Stack*) malloc(sizeof(T_Stack));
  if (s == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  s->top_index = -1;
  s->capacity = 100;
  s->item = malloc(s->capacity * sizeof(Tetrahedron*));
  if (s->item == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  return s;
}

int is_empty_t_stack (T_Stack *s) {
  return ( s->top_index == -1 );
}

Tetrahedron *top_t (T_Stack * s) {
  if(!is_empty_t_stack(s)) {
    return s->item [s->top_index];
  }
  else {
    fprintf(stderr,"ERROR top called on empty T_Stack");
    exit(-1);
  }
}

void push_t (Tetrahedron *t, T_Stack * s) {
  int i = s -> top_index + 1;
  if ( i >= s -> capacity ) {
    s->capacity *= 2;
    s->item = realloc(s->item, s->capacity * sizeof(Tetrahedron*));
  }
  s->item[i] = t;
  s->top_index = i;
}

void pop_t (T_Stack * s) {
  -- (s -> top_index);
}

void free_t_stack(T_Stack *s) {
  free(s->item);
  free(s);
}

//square the double d
double sq (double d) {
  return d * d;
}

//Checks if the point (e0,e1,e2) lies inside the positively oriented
//Tetrahedron *t's circumsphere.
int in_sphere(Tetrahedron *t, double e0, double e1, double e2) {
  double a[4], b[4], c[4], d[4];
  a[0] = p0(t->v[0]) - e0;
  a[1] = p1(t->v[0]) - e1;
  a[2] = p2(t->v[0]) - e2;
  a[3] = sq(a[0]) + sq(a[1]) + sq(a[2]);
  b[0] = p0(t->v[1]) - e0;
  b[1] = p1(t->v[1]) - e1;
  b[2] = p2(t->v[1]) - e2;
  b[3] = sq(b[0]) + sq(b[1]) + sq(b[2]);
  c[0] = p0(t->v[2]) - e0;
  c[1] = p1(t->v[2]) - e1;
  c[2] = p2(t->v[2]) - e2;
  c[3] = sq(c[0]) + sq(c[1]) + sq(c[2]);
  d[0] = p0(t->v[3]) - e0;
  d[1] = p1(t->v[3]) - e1;
  d[2] = p2(t->v[3]) - e2;
  d[3] = sq(d[0]) + sq(d[1]) + sq(d[2]);

  if (determinant_4(a, b, c, d) <= 0)
    return 0;
  else
    return 1;
}

//Checks if face f is shared with a tetrahedron in list.
//The search starts at node in list, and does backwards search, then forawrds.
int shared_face_in_graph(Face e, DLL_NODE *node, DLL *list) {
  DLL_NODE *temp;

  temp = node->prev;
  while (temp != NULL) {
    if (equal(e, temp->data->f[0])) return 1;
    else if (equal(e, temp->data->f[1])) return 1;
    else if (equal(e, temp->data->f[2])) return 1;
    else if (equal(e, temp->data->f[3])) return 1;
    temp = temp->prev;
  }

  temp = node->next;
  while (temp != NULL) {
    if (equal(e, temp->data->f[0])) return 1;
    else if (equal(e, temp->data->f[1])) return 1;
    else if (equal(e, temp->data->f[2])) return 1;
    else if (equal(e, temp->data->f[3])) return 1;
    temp = temp->next;
  }
  return 0;
}

/*****************************************************************************
Recursively checks the neighbours of Tetrahedron *t. If a neighbour is bad it is
added to DLL *bad and checked itself. Otherwise neighbour is marked as checked.
At each step, tetrahedrons are ignored that have a checked value equal to run.
*****************************************************************************/
void check_neighbours(Tetrahedron *t, int run, DLL *bad) {
  DLL_NODE *node;
  Tetrahedron *nbhr;
  int i;

  t->checked = run; //Don't look at this tetrahedron again this step.
  for(i = 0; i < 4; ++i) {
    nbhr = t->adj[i];
    //Check if the neighbour is non NULL
    if(nbhr != NULL) {
      //Only check each tetrahedron once
      if(nbhr->checked < run) {
        if (in_sphere(nbhr, p0(run), p1(run), p2(run))) {
          node = find_node(nbhr, del);
          add_to_list(nbhr, bad);
          remove_node(node, del);
          check_neighbours(nbhr, run, bad);
        }
        else {
          nbhr->checked = run;
        }
      }
    }
  }
}

//Checks if *t has face f belonging to nbhr, and if so updates *t's adjacency
int find_matching_face(Tetrahedron *t, Face f, Tetrahedron *nbhr) {
  if (equal(t->f[0], f)) {
    t->adj[0] = nbhr;
    return 1;
  }
  if (equal(t->f[1], f)) {
    t->adj[1] = nbhr;
    return 1;
  }
  if (equal(t->f[2], f)) {
    t->adj[2] = nbhr;
    return 1;
  }
  if (equal(t->f[3], f)) {
    t->adj[3] = nbhr;
    return 1;
  }
  fprintf(stderr,"Found no match for a face in a tetrahedron\n");
  exit(-1);
  return 0;
}

/*****************************************************************************
Checks if face f is shared by any tetrahedron in a search starting at node and
moving backwards. If it is, the tetrahedron pointed to by node has
adj[face_no] updated with the found tetrahedron.
*****************************************************************************/
void find_adjacent_tetra_to_face(Face f, DLL_NODE *node, int face_no) {
  DLL_NODE *comp = node->prev; //Why can't this be NULL?

  if (node->data->adj[face_no] == NULL) {
    while(comp != NULL) {
        if (equal(f, comp->data->f[0])) {
          comp->data->adj[0] = node->data;
          node->data->adj[face_no] = comp->data;
          break;
        }
        else if (equal(f, comp->data->f[1])) {
          comp->data->adj[1] = node->data;
          node->data->adj[face_no] = comp->data;
          break;
        }
        else if (equal(f, comp->data->f[2])) {
          comp->data->adj[2] = node->data;
          node->data->adj[face_no] = comp->data;
          break;
        }
        else if (equal(f, comp->data->f[3])) {
          comp->data->adj[3] = node->data;
          node->data->adj[face_no] = comp->data;
          break;
        }
      comp = comp->prev;
    }
  }
}

//Finds all adjacencies moving backwards in a list starting at start
void find_adjacencies(DLL_NODE *start) {
  Face face;
  DLL_NODE *node;

  node = start;
  while (node != NULL) {
    //check face 1
    face = node->data->f[0];
    find_adjacent_tetra_to_face(face, node, 0);
    //check face 2
    face = node->data->f[1];
    find_adjacent_tetra_to_face(face, node, 1);
    //check face 3
    face = node->data->f[2];
    find_adjacent_tetra_to_face(face, node, 2);
    //check face 4
    face = node->data->f[3];
    find_adjacent_tetra_to_face(face, node, 3);

    node = node->prev;
  }
}

//Make all Tetrahedrons which *t points to point to NULL instead of back to *t
void delete_ties(Tetrahedron *t) {
  int i, j;
  for(i = 0; i < 4; ++i) {
    if(t->adj[i] != NULL) {
      for(j = 0; j < 4; ++j) {
        if(t->adj[i]->adj[j] == t) {
          t->adj[i]->adj[j] = NULL;
          break;
        }
      }
    }
  }
}

//Produces the Delaunay tetrahedrilisation of our points in p
void bowyer_watson() {
  int i, j;
  int index, shared_face, found_bad;
  Face f;
  Tetrahedron *t1, *t2;
  DLL *bad = make_list();
  DLL_NODE *node, *temp;
  Stack *polyhedron = make_stack(); //The Delaunay cavity
  //border_tetras will hold the good tetrahedrons on the border of the cavity
  T_Stack *border_tetras = make_t_stack();

  //Add super Tetrahedron vertices to array of points
  index = 3 * num_points - 1;
  p[++index] = 0;       p[++index] = 0;        p[++index] = 100000; //Vertice 1
  p[++index] = -100000; p[++index] = -100000;  p[++index] = -100000;//Vertice 2
  p[++index] = 100000;  p[++index] = -100000;  p[++index] = -100000;//Vertice 3
  p[++index] = 0;       p[++index] = 100000;   p[++index] = -100000;//Vertice 4

  Tetrahedron *big_t =
  make_tetrahedron(num_points, num_points + 1, num_points + 2, num_points + 3);
  add_to_list(big_t, del);

  for (i = 0; i < num_points; ++i) {
    empty_dll(bad); //reset the bad tetrahedrons
    node = del->first;
    found_bad = 0;
    while (!found_bad) { //Find one bad tetrahedron
      if (node == NULL) {
        fprintf(stderr, "%s\n", "ERROR: found no bad tetrahedron");
        exit(-1);
      }
      if (in_sphere(node->data, p0(i), p1(i), p2(i))) {
        add_to_list(node->data, bad);
        remove_node(node, del);
        found_bad = 1;
      }
      else node = node->next;
    }

    check_neighbours(bad->first->data, i, bad); //Find all bad tetrahedrons
    node = bad->first;
    while(node != NULL) {
      for(j = 0; j < 4; ++j) {
        f = node->data->f[j];
        shared_face = shared_face_in_graph(f, node, bad);
        if(!shared_face) {//create the Delaunay cavity
          push(f, polyhedron);
          push_t(node->data->adj[j], border_tetras);
        }
      }
      node = node->next;
    }
    //Create the new tetrahedrons and update adjacencies
    //with the good tetrahedrons on the border of the cavity.
    f = top(polyhedron);
    add_to_list(make_tetrahedron(f.v[0], f.v[1], f.v[2], i), del);
    node = del->first; //different line - marks start of new tetrahedrons
    t2 = top_t(border_tetras);
    if(t2 != NULL) {
      t1 = del->first->data;
      find_matching_face(t1, f, t2);
      find_matching_face(t2, f, t1);
    }
    pop_t(border_tetras);
    pop(polyhedron);
    //inside the while loop is a repeat of the above without the different line
    while (!is_empty_stack(polyhedron)) {
      f = top(polyhedron);
      add_to_list(make_tetrahedron(f.v[0], f.v[1], f.v[2], i), del);
      t2 = top_t(border_tetras);
      if(t2 != NULL) {
        t1 = del->first->data;
        find_matching_face(t1, f, t2);
        find_matching_face(t2, f, t1);
      }
    pop_t(border_tetras);
    pop(polyhedron);
    }
    find_adjacencies(node); //Fill in adjacencies between new tetrahedrons
  }
  node = del->first;
  while (node != NULL) {
    //Remove tetrahedrons which intersect the super tetrahedron
    if(shares_vertex_supert(node->data, num_points) == 1) {
      temp = node;
      node = node->next;
      delete_ties(temp->data);
      free(temp->data);
      remove_node(temp, del);
    }
    else {
      node = node->next;
    }
  }
  //clean up
  empty_dll(bad);
  free(bad);
  free_stack(polyhedron);
  free_t_stack(border_tetras);
}

//returns the Euclidean norm squared of input
double norm_sq(double *input) {
  return sq(input[0]) + sq(input[1]) + sq(input[2]);
}

//subtracts b from a and stores in r
void vector_subtraction(double a1, double a2, double a3,
                         double b1, double b2, double b3, double *r) {
  r[0] = a1 - b1;
  r[1] = a2 - b2;
  r[2] = a3 - b3;
}

//crosses u with v and stores in r
void cross_product(double *u, double *v, double *r) {
  r[0] = determinant_2(u[1], u[2], v[1], v[2]);
  r[1] = -1 * determinant_2(u[0], u[2], v[0], v[2]);
  r[2] = determinant_2(u[0], u[1], v[0], v[1]);
}

//Multiplies the vector a by the scalar k, storing back in a
void scalar_mult(double k, double *a) {
  a[0] *= k; a[1] *= k; a[2] *= k;
}

//Finds the circumcentre of *tet
void circumcentre_sphere(Tetrahedron *tet) {
  double r;
  double volume; //Will hold scaled volume
  double t[3], u[3], v[3]; //Want to translate tet->v[3] to the origin
  double uxv[3], vxt[3], txu[3]; //Will hold cross products
  vector_subtraction(p0(tet->v[0]), p1(tet->v[0]), p2(tet->v[0]),
                     p0(tet->v[3]), p1(tet->v[3]), p2(tet->v[3]), t);
  vector_subtraction(p0(tet->v[1]), p1(tet->v[1]), p2(tet->v[1]),
                     p0(tet->v[3]), p1(tet->v[3]), p2(tet->v[3]), u);
  vector_subtraction(p0(tet->v[2]), p1(tet->v[2]), p2(tet->v[2]),
                     p0(tet->v[3]), p1(tet->v[3]), p2(tet->v[3]), v);
  //norms are really the norms squared
  double t_norm = norm_sq(t);
  double u_norm = norm_sq(u);
  double v_norm = norm_sq(v);
  cross_product(u, v, uxv);
  cross_product(v, t, vxt);
  cross_product(t, u, txu);
  volume = tetrahedron_determinant(tet);
  r = (t_norm * uxv[0]) + (u_norm * vxt[0]) + (v_norm * txu[0]);
  tet->circum[0] = p0(tet->v[3]) + (r / (2 * volume));
  r = (t_norm * uxv[1]) + (u_norm * vxt[1]) + (v_norm * txu[1]);
  tet->circum[1] = p1(tet->v[3]) + (r / (2 * volume));
  r = (t_norm * uxv[2]) + (u_norm * vxt[2]) + (v_norm * txu[2]);
  tet->circum[2] = p2(tet->v[3]) + (r / (2 * volume));
}
```

```c

//Computes the circumcentre of face t, storing in cc - is coplanar with t
void circumcentre_circle(Face t, double *cc) {
  double a[3], b[3], c[3]; //Want to translate t->v[2] to the origin
  double axb[3], cxaxb[3]; //Will hold cross products
  vector_subtraction(p0(t.v[0]), p1(t.v[0]), p2(t.v[0]),
                     p0(t.v[2]), p1(t.v[2]), p2(t.v[2]), a);
  vector_subtraction(p0(t.v[1]), p1(t.v[1]), p2(t.v[1]),
                     p0(t.v[2]), p1(t.v[2]), p2(t.v[2]), b);
  cross_product(a, b, axb);
  //Norms are really the norm squared
  double a_norm = norm_sq(a);
  double b_norm = norm_sq(b);
  double axb_norm = norm_sq(axb);
  scalar_mult(a_norm, b);
  scalar_mult(b_norm, a);
  vector_subtraction(b[0], b[1], b[2], a[0], a[1], a[2] ,c);
  cross_product(c, axb, cxaxb); // read cxaxb as c x (a x b)
  cc[0] = (cxaxb[0] / (2 * axb_norm)) + p0(t.v[2]);
  cc[1] = (cxaxb[1] / (2 * axb_norm)) + p1(t.v[2]);
  cc[2] = (cxaxb[2] / (2 * axb_norm)) + p2(t.v[2]);
}

//Finds all edges containing v in del, storing in edges and a pointer to
//a tetrahedron containing that edge is stored in tetras.
//Returns the number of edges found sharing v.
int find_all_edges(int v, Edge *edges, Tetrahedron **tetras, int *capacity) {
  DLL_NODE *node = del->first;
  int i, j;
  int n;
  int num_edges = 0;
  Tetrahedron *t;
  Edge e;
  int new_edge;
  while (node != NULL) {
    t = node->data;
    if((t->v[0] == v) || (t->v[1] == v) || (t->v[2] == v) || (t->v[3] == v)) {
      if(t->v[0] == v) n = 0;
      else if(t->v[1] == v) n = 1;
      else if(t->v[2] == v) n = 2;
      else n = 3;
      e.v[0] = t->v[n];
      for (j = 1; j < 4; ++j) {
        new_edge = 1;
        e.v[1] = t->v[(n + j) % 4];
        for (i = 0; i < num_edges; ++i)
          if(equal_edges(edges[i], e)) {
            new_edge = 0;
          }
        if(new_edge) {
          if ((num_edges + 1) > *capacity) {
            *capacity *= 2;
            if(realloc(edges, *capacity * sizeof(Edge)) == NULL) {
              fprintf(stderr, "ERROR: realloc failed\n");
              exit(-1);
            }
            if(realloc(tetras, *capacity * sizeof(Tetrahedron*)) == NULL) {
              fprintf(stderr, "ERROR: realloc failed\n");
              exit(-1);
            }
          }
          tetras[num_edges] = t;
          edges[num_edges++] = e;
        }
      }
    }
    node = node->next;
  }
  return num_edges;
}

//find another point on the line ab, in direction ab, storing in r
void point_on_line(double *a, double *b, double *r) {
  double min;
  double d, e, f;
  int i;
  if(fabs(b[0] - a[0]) == 0) d = 100000;
  else d = fabs(b[0] - a[0]);
  if(fabs(b[1] - a[1]) == 0) e = 100000;
  else e = fabs(b[1] - a[1]);
  if(fabs(b[2] - a[2]) == 0) f = 100000;
  else f = fabs(b[2] - a[2]);
  min = d < e ? d : e;
  min = min < f ? min : f;
  for (i = 0; i < 3; ++i) {
    //find vector between a and b, scale it and add back a
    r[i] = (b[i] - a[i]) * (200 / min) + a[i];
  }
}

//r will contain the indices of the faces in *t containg e
void faces_with_edge(Edge e, Tetrahedron *t, int *r) {
  int i, j;
  Face f;
  int check;
  int count = 0;
  for (j = 0; j < 4; ++j) {
    f = t->f[j];
    check = 0;
    for (i = 0; i < 3; ++i) {
      if((f.v[i] == e.v[0]) || (f.v[i] == e.v[1])) ++check;
    }
    if (check == 2) {
      r[count] = j;
      ++count;
      if(count == 2) break;
    }
  }
}

//Will see if point d forms a positively oriented tetrahedron with triangle f
double orientation_check(Face f, double *d) {
  double a[3], b[3], c[3];
  a[0] = p0(f.v[0]) - d[0];
  a[1] = p1(f.v[0]) - d[1];
  a[2] = p2(f.v[0]) - d[2];
  b[0] = p0(f.v[1]) - d[0];
  b[1] = p1(f.v[1]) - d[1];
  b[2] = p2(f.v[1]) - d[2];
  c[0] = p0(f.v[2]) - d[0];
  c[1] = p1(f.v[2]) - d[1];
  c[2] = p2(f.v[2]) - d[2];
  return determinant_3(a, b, c);
}

//Finds a point on the unbounded voronoi edge formed by Face f
//Which belongs to a tetrahedron with circumcentre a, storing the point in pg
void compute_unbounded_edge(Face f, double *a, Polygon_3D *pg) {
  double cc[3], temp[3];
  double check;
  circumcentre_circle(f, cc);
  check = orientation_check(f, a);
  if (check == 0) {
    fprintf(stderr,"Error: circumcentre lies on face of tetrahedron\n");
    fprintf(stderr,"Circumcentre is %lf %lf %lf\n", a[0], a[1], a[2]);
    fprintf(stderr,"Lying on face:\n");
    fprintf(stderr, "%lf %lf %lf\n", p0(f.v[0]), p1(f.v[0]), p2(f.v[0]));
    fprintf(stderr, "%lf %lf %lf\n", p0(f.v[1]), p1(f.v[1]), p2(f.v[1]));
    fprintf(stderr, "%lf %lf %lf\n", p0(f.v[2]), p1(f.v[2]), p2(f.v[2]));
    exit(-1);
  }
  //Depending on the orientation of the tetrahedron formed by f and a,
  //the necessary unbounded edge will point in different directions
  if (check > 0) point_on_line(a, cc, temp);
  else point_on_line(cc, a, temp);
  add_to_polygon(temp, pg);
}

//Copies pg1 into pg2, in reverse order of points.
void copy_pg_reverse(Polygon_3D *pg1, Polygon_3D *pg2) {
  int i;
  double a[3];
  for(i = ((pg1->num_items / 3) - 1); i >= 0; --i) {
    a[0] = pg1->v[3 * i];
    a[1] = pg1->v[3 * i + 1];
    a[2] = pg1->v[3 * i + 2];
    add_to_polygon(a, pg2);
  }
}

//Copies pg1 into pg2
void copy_pg(Polygon_3D *pg1, Polygon_3D *pg2) {
  int i;
  double a[3];
  for(i = 0; i < (pg1->num_items/ 3); ++i) {
    a[0] = pg1->v[3 * i];
    a[1] = pg1->v[3 * i + 1];
    a[2] = pg1->v[3 * i + 2];
    add_to_polygon(a, pg2);
  }
}

//Finds the Voronoi face for Delaunay edge e in tetrahedron *start
//This is associated to the polyhedron dual to the vertex labelled vertex.
//The Voronoi face will the faceindexed by face_num in the polyhedron
void voronoi_face(Tetrahedron *start, Edge e,
                  int vertex, int face_num, Polygon_3D *temp_pg) {
  Tetrahedron *current, *prev, *next;
  //Below will hold the 2 faces of a tetrahedron that share a certain edge.
  int faces[2], start_faces[2];
  int i = 0;
  Polygon_3D *pg;
  Face f;
  int done = 0;

  //This is the face of the polygon that we are working with:
  pg = voro[vertex]->faces[face_num];
  if(start == NULL) {
    fprintf(stderr, "ERROR: started with NULL tetrahedron\n");
    exit(-1);
  }
  temp_pg->num_items = 0;
  add_to_polygon(start->circum, temp_pg);
  faces_with_edge(e, start, start_faces);
  while((i < 2) && !done) {//i == 0 first direction, i == 1 second direction
    prev = start;
    if(start->adj[start_faces[i]] != NULL) {
      next = start->adj[start_faces[i]];
      while(next != start) {//Not returned to starting point
        current = next;
        if(i == 1) add_to_polygon(current->circum, pg);
        else add_to_polygon(current->circum, temp_pg);
        faces_with_edge(e, current, faces);
        if (current->adj[faces[0]] != prev) {
          if (current->adj[faces[0]] == NULL) {
            f = current->f[faces[0]];
            if(i == 1) compute_unbounded_edge(f, current->circum, pg);
            else {
              compute_unbounded_edge(f, current->circum, temp_pg);
              copy_pg_reverse(temp_pg, pg);
            }
            voro[vertex]->unbounded = 1;
            break;
          }
          next = current->adj[faces[0]];
          prev = current;
        }
        else if (current->adj[faces[1]] == prev) {
          fprintf(stderr, "ERROR: incorrect adjacency\n");
          fprintf(stderr, "took face %d to start\n", i);
          fprintf(stderr, "%p and %p should not be adjacent",
                          prev, current->adj[faces[1]]);
          exit(-1);
        }
        else {
          if (current->adj[faces[1]] == NULL) {
            f = current->f[faces[1]];
            if(i == 1) compute_unbounded_edge(f, current->circum, pg);
            else {
              compute_unbounded_edge(f, current->circum, temp_pg);
              copy_pg_reverse(temp_pg, pg);
            }
            voro[vertex]->unbounded = 1;
            break;
          }
          next = current->adj[faces[1]];
          prev = current;
        }
      }
      if(next == start) done = 1;
    }
    else {
      f = start->f[start_faces[i]];
      if(i == 1) compute_unbounded_edge(f, start->circum, pg);
      else {
        compute_unbounded_edge(f, start->circum, temp_pg);
        copy_pg_reverse(temp_pg, pg);
      }
      voro[vertex]->unbounded = 1;
    }
    ++i;
  }
  if(done) copy_pg(temp_pg, pg);
}

//Computes the Voronoi diagram of input point set
void voronoi() {
  int i, j;
  int num_edges;
  DLL_NODE *node;
  int capacity = 200;
  Polygon_3D *temp_pg = make_polygon(3000);
  //edges stores each edge e with i as a vertice, and tetras stores a
  //tetrahedron containing e, for each aforementioned edge.
  Tetrahedron **tetras = malloc(capacity * sizeof(Tetrahedron*));
  Edge *edges = malloc(capacity * sizeof(Edge));
  if ((edges == NULL) || (tetras == NULL)){
    fprintf(stderr, "%s\n", "ERROR: Malloc failed");
    exit(-1);
  }

  node = del->first;
  while (node != NULL) { //Compute the circumcentres
    circumcentre_sphere(node->data);
    node = node->next;
  }

  voro = malloc(num_points * sizeof(Polyhedron*));
  if (voro == NULL){
    fprintf(stderr, "%s\n", "ERROR: Malloc failed");
    exit(-1);
  }

  for(i = 0; i < num_points; ++i) { //Compute Voronoi region
    num_edges = find_all_edges(i, edges, tetras, &capacity);
    voro[i] = make_polyhedron(num_edges);
    for(j = 0; j < num_edges; ++j) { //Compute Voronoi face
      voronoi_face(tetras[j], edges[j], i, j, temp_pg);
    }
  }
  free(temp_pg);
}

//Prints the input points to a file
void print_points() {
  int i;
  FILE *file;
  if ((file = fopen("Logs/Points_3D.txt","w")) == NULL) {
    fprintf(stderr, "File not openable \n");
    exit(-1);
  }
  fprintf(file, "%d\n", num_points);
  for (i = 0; i < num_points; ++i) {
    fprintf(file, "%lf %lf %lf\n", p0(i), p1(i), p2(i));
  }
  if(fclose(file) == EOF) {
    fprintf(stderr, "Couldn't close file");
    exit(-1);
  }
}

//Prints the voronoi diagram to a file
void print_voronoi() {
  int i, j, k;
  FILE *file;
  if ((file = fopen("Logs/Voro_3D.txt","w")) == NULL) {
    fprintf(stderr, "File not openable \n");
    exit(-1);
  }
  Polygon_3D * pg;
  for(i = 0; i < num_points; ++i) {
    if(voro[i]->unbounded == 0) fprintf(file, "*****Bounded ");
    else fprintf(file, "*****Unbounded ");
    fprintf(file, "Polyhedron %d follows:*****\n", i);
    for(j = 0; j < voro[i]->num_faces; ++j) {
      pg = voro[i]->faces[j];
      fprintf(file, "Face number %d:\n", j);
      for(k = 0; k < (pg->num_items / 3); ++k) {
        fprintf(file,"%lf %lf %lf\n",
                pg->v[3 * k], pg->v[3 * k + 1], pg->v[3 * k + 2]);
      }
      fprintf(file, "\n");
    }
  }
  if(fclose(file) == EOF) {
    fprintf(stderr, "Couldn't close file");
    exit(-1);
  };
}

//Finds a bounding box for the input points
void bounding_box() {
  int i;
  bmin[0] = bmin[1] = bmin[2] = 0;
  bmax[0] = bmax[1] = bmax[2] = 0;
  for (i = 0; i < num_points; ++i) {
    if (bmin[0] > p0(i)) bmin[0] = p0(i);
    else if (bmax[0] < p0(i)) bmax[0] = p0(i);
    if (bmin[1] > p1(i)) bmin[1] = p1(i);
    else if (bmax[1] < p1(i)) bmax[1] = p1(i);
    if (bmin[2] > p2(i)) bmin[2] = p2(i);
    else if (bmax[2] < p2(i)) bmax[2] = p2(i);
  }
  bmin[0] -= 0.5; bmax[0] += 0.5;
  bmin[1] -= 0.5; bmax[1] += 0.5;
  bmin[2] = -fabs(bmax[2]) - 1.5; bmax[2] = fabs(bmin[2]) + 0.5;
  glMatrixMode(GL_PROJECTION);  // To operate on the Projection matrix
  glLoadIdentity();
  glOrtho(bmin[0], bmax[0],
          bmin[1], bmax[1],
          bmin[2], bmax[2]);
}

//This draws the Delaunay tetrahedrilisation with colours.
void draw_tetrahedrons() {
  int i = 0;
  int j;
  DLL_NODE *node;
  Face f;

  glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
  glBegin(GL_TRIANGLES);
  node = del->first;
  while (node != NULL) {
    glColor4d(color[3 * i], color[3 * i + 1], color[3 * i + 2], 0.5);
    for (j = 0; j < 4; ++j) {
      f = node->data->f[j];
      glVertex3d(p0(f.v[0]), p1(f.v[0]), p2(f.v[0]));
      glVertex3d(p0(f.v[1]), p1(f.v[1]), p2(f.v[1]));
      glVertex3d(p0(f.v[2]), p1(f.v[2]), p2(f.v[2]));
    }
    node = node->next;
    ++i;
    }
  glEnd();
}

//This draws a blue mesh of the delaunay tetrahedrilisation
void draw_tetrahedron_mesh() {
  DLL_NODE *node;
  int i;
  double a, b, c;
  Face f;

  glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
  glBegin(GL_TRIANGLES);
  a = 0.1, b = 0.3, c = 0.6;
  node = del->first;
  while (node != NULL) {
    glColor4d(a, b, c, 0.5);
    for (i = 0; i < 4; ++i) {
      f = node->data->f[i];
      glVertex3d(p0(f.v[0]), p1(f.v[0]), p2(f.v[0]));
      glVertex3d(p0(f.v[1]), p1(f.v[1]), p2(f.v[1]));
      glVertex3d(p0(f.v[2]), p1(f.v[2]), p2(f.v[2]));
    }
    node = node->next;
  }
  glEnd();
}

//This draws the original points in yellow
void draw_points(int color_var, int cull_on) {
  int i;

  glBegin(GL_POINTS);
  if(color_var == 0)
    glColor3d(1.0,1.0,1.0);
  else
    glColor3d(1.0,1.0,0.0);
  for (i = 0; i < num_points; ++i) {
    if(!(cull_on && voro[i]->unbounded))
      glVertex3d(p0(i), p1(i), p2(i));
  }
  glEnd();
}

//Draws the Voronoi diagram in colour. If cull_on unbounded regions are removed.
void draw_voronoi(int cull_on) {
  int i, j, k;
  Polygon_3D * pg;
  for(i = 0; i < num_points; ++i) {
    if(!(cull_on && voro[i]->unbounded)) {
      glColor4d(color[3 * i], color[3 * i + 1], color[3 * i + 2], 0.5);
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
      for(j = 0; j < voro[i]->num_faces; ++j) {
        pg = voro[i]->faces[j];
        glBegin(GL_POLYGON);
        for(k = 0; k < (pg->num_items / 3); ++k) {
          glVertex3d(pg->v[3 * k], pg->v[3 * k + 1], pg->v[3 * k + 2]);
        }
        glEnd();
      }
    }
  }
}

//Draws the Voronoi diagram in mesh. If cull_on unbounded regions are removed.
void draw_voronoi_mesh(int cull_on) {
  int i, j, k;
  Polygon_3D * pg;
  for(i = 0; i < num_points; ++i) {
    if(!(cull_on && voro[i]->unbounded)){
      glColor3d(1.0, 1.0, 1.0);
      glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
      for(j = 0; j < voro[i]->num_faces; ++j) {
        pg = voro[i]->faces[j];
        glBegin(GL_POLYGON);
        for(k = 0; k < (pg->num_items / 3); ++k) {
          glVertex3d(pg->v[3 * k], pg->v[3 * k + 1], pg->v[3 * k + 2]);
        }
        glEnd();
      }
    }
  }
}

//Below are functions for moving the figure around
void rotate_x_up() {
  r_x += 2.0;
  fmod(r_x, 360);
}

void rotate_x_down() {
  r_x -= 2.0;
  fmod(r_x, 360);
}

void rotate_y_up() {
  r_y += 2.0;
  fmod(r_y, 360);
}

void rotate_y_down() {
  r_y -= 2.0;
  fmod(r_y, 360);
}

void zoom_in() {
  zoom *= 1.05;
}

void zoom_out() {
  zoom /= 1.05;
}
```

```

void tumbling() {
  if (tumble_on) {
    tumble += 1.2;
    fmod(tumble, 360);
    #ifdef _WIN32
    Sleep(50);
    #else
    sleep(0.03);
    #endif
    glutPostRedisplay();
  }
}

//Displays the results graphically
void display() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  //glTranslated(0.0, 0.0, -max -4.0);
  glRotated(tumble, 0.0, 1.0, 0.0);
  glRotated(tumble, 1.0, 1.0, 1.0);
  glRotated(r_x, 1.0, 0.0, 0.0);
  glRotated(r_y, 0.0, 1.0, 0.0);
  glScaled(zoom, zoom, zoom);

  if(p_on)
    draw_points(0, cull_on);
  if (state == 0) draw_points(0, 0);
  else if (state == 1) draw_tetrahedrons();
  else if (state == 2)
    draw_voronoi(cull_on);
  else if (state == 3) {
    if(d_on)
      draw_tetrahedron_mesh();
    if(v_on)
      draw_voronoi_mesh(cull_on);
  }
  glutSwapBuffers();
  tumbling();
}

//Initiliase our display settings
void init() {
  glMatrixMode(GL_PROJECTION);
  glClearColor(0.0f, 0.0f, 0.0f, 1.0f); //Black background
  glPointSize(3.00);
  //Enable Depth testing:
  glClearDepth(1.0f);
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LEQUAL);
  //Making things look nicer
  glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
  glHint(GL_POLYGON_SMOOTH_HINT, GL_NICEST);
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}

//reshape window on Initiliase, and reshape.
void reshape(GLsizei width, GLsizei height) {
  glClearColor ( 0,0,0,0 );
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  // Set the viewport to cover the new window
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);  // To operate on the Projection matrix
  glLoadIdentity();             // Reset
  //Establish clipping planes.
  glOrtho(bmin[0], bmax[0], bmin[1], bmax[1], -bmin[2], -bmax[2]);
}

void new_clip() {
  glClearColor(0,0,0,0);
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glutSwapBuffers();
  printf("Current clip planes are:\n");
  printf("x: left %lf, right %lf\n", bmin[0], bmax[0]);
  printf("y: bottom %lf, top %lf\n", bmin[1], bmax[1]);
  printf("z: near %lf, far %lf, negative z is into screen\n",
         bmin[2], bmax[2]);
  printf("Please define new clipping planes\n");
  printf("Please enter your new x clip co-ordinates as: left right\n");
  scanf("%lf %lf", &bmin[0], &bmax[0]);
  printf("Please enter your new y clip co-ordinates as: bottom top\n");
  scanf("%lf %lf", &bmin[1], &bmax[1]);
  printf("Please enter your new z clip co-ordinates as: near far\n");
  scanf("%lf %lf", &bmin[2], &bmax[2]);
  glMatrixMode(GL_PROJECTION);  // To operate on the Projection matrix
  glLoadIdentity();
  glOrtho(bmin[0], bmax[0], bmin[1], bmax[1], -bmin[2], -bmax[2]);
}

//A looping keyboard function
void kbd ( unsigned char key, int x, int y ) {
  switch ( key ) {
    case 'w':
      rotate_x_down();
      glutPostRedisplay();
      break;

    case 's':
      rotate_x_up();
      glutPostRedisplay();
      break;

    case 'a':
      rotate_y_down();
      glutPostRedisplay();
      break;

    case 'd':
      rotate_y_up();
      glutPostRedisplay();
      break;

    case 'q':
      zoom_in();
      glutPostRedisplay();
      break;

    case 'e':
      zoom_out();
      glutPostRedisplay();
      break;

    case 27: //ESC key
      exit(0);
      break;

    case 'p':
      state = 0;
      glutPostRedisplay();
      break;

    case 't':
      state = 1;
      glutPostRedisplay();
      break;

    case 'v':
      state = 2;
      glutPostRedisplay();
      break;

    case 'c':
      state = 3;
      glutPostRedisplay();
      break;

    case 'r':
      tumble_on = 1 - tumble_on;
      glutPostRedisplay();
      break;

    case 'b':
      cull_on = 1 - cull_on;
      glutPostRedisplay();
      break;

    case 'P':
      p_on = 1 - p_on;
      glutPostRedisplay();
      break;

    case 'T':
      d_on = 1 - d_on;
      glutPostRedisplay();
      break;

    case 'V':
      v_on = 1 - v_on;
      glutPostRedisplay();
      break;

    case 'n':
      new_clip();
      glutPostRedisplay();
      break;

    case 'N':
      bounding_box();
      glutPostRedisplay();
      break;

    default:
      fprintf(stderr,"key (%c:%x) is not bound\n", key, key);
      fprintf(stderr,"Bound keys are ESC to quit,"
                     " p for points only, P to draw points over objects\n"
                     "t for tetrahedrilisation - Delaunay, "
                     "v for Voronoi diagram \n"
                     "n for new clip planes, N to reset clip planes\n"
                     "b to cull unbounded polyhedrons, r for random rotation\n"
                     "Camera movement with w, a, s, d. Zoom with q, e\n"
                     "c to enter multiple drawings mode - In this mode:\n"
                     "V - Voronoi, T - Delaunay\n");
      break;
  }
}
```

```c

//Reads the input points either from stdin or reads them from a file
//if from_file, then the file read from string input
void read_points(int from_file, char *input) {
  int i;
  FILE *file;
  if(!from_file) {
    printf("Enter how many points you want to enter\n");
    scanf("%d", &num_points);
    p = calloc(3 * (num_points + 4), sizeof(double));

    printf("Please enter your 3D points separated by spaces\n");

    for (i = 0; i < num_points; ++i)
      scanf("%lf %lf %lf", &p[3 * i], &p[3 * i + 1], &p[3 * i + 2]);
  }

  else {
    if ((file = fopen(input, "r")) == NULL) {
      fprintf(stderr, "File not openable \n");
      exit(-1);
    }
    fscanf(file, "%d\n", & num_points);
    fprintf(stderr, "number of points is %d\n", num_points);
    p = malloc(3 * (num_points + 4) * sizeof(double));
    if (p == NULL) {
      fprintf(stderr, "ERROR: malloc failed\n");
      exit(-1);
    }

    for (i = 0; i < num_points; ++i )
        fscanf(file, "%lf %lf %lf\n", &p[3 * i], &p[3 * i + 1], &p[3 * i + 2]);

    if(fclose(file) == EOF){
      fprintf(stderr,"Couldn't close the file\n");
      exit(-1);
    }
  }
}

//Produces num_p random input points
void randomise(int num_p) {
  int i;
  int a, b, c;
  int divide;

  divide = ceil(0.01 * (num_p/2.0));
  if(divide > 10) divide = 10;
  p = malloc(3 * (num_points + 4) * sizeof(double));
  if (p == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  for(i = 0; i < num_p; ++i) {
    a = rand() % 2;
    b = rand() % 2;
    c = rand() % 2;
    if(a == 0)
      a = -1;
    if(b == 0)
      b = -1;
    if(c == 0)
      c = -1;
    p[3 * i] = a * (double)(rand()) / (RAND_MAX/divide);
    p[3 * i + 1] = b * (double)(rand()) / (RAND_MAX/divide);
    p[3 * i + 2] = c * (double)(rand()) / (RAND_MAX/divide);
  }
}

//Allocates memory to globals and calls funtions
int main(int argc, char **argv) {
  clock_t ti;
  int i;
  int max;
  int from_file = 0;
  int random_on = 0;
  time_t t;
  int num_tetras;

  #ifdef _WIN32
    srand((unsigned)time(&t));
  //srand((long)1);
  #else
    srand48((unsigned)time(&t));
  #endif

  //check how prog should be run
  for (i = 0; i < argc; ++i) {
    if(strcmp(argv[i], "-file") == 0) {
      //Read from a file.
      from_file  = 1;
    }
    if(strcmp(argv[i], "-debug") == 0) {
      //Turn on logs and stderr messages.
      debug = 1;
    }
    if(strcmp(argv[i], "-random") == 0) {
      //Create a random set of points.
      random_on = 1;
    }
    if(strcmp(argv[i], "-help") == 0) {
      fprintf(stderr,
              "This is a program to compute and display the following:\n"
              "The Voronoi diagram and"
              " Delaunay tetrahedrilisation of 3D points.\n"
              "Run the program to enter points through stdin.\n"
              "Run with \"-file\" {filename} to give the program a file. File input format is:\n"
              "3\n0.0 0.5 10\n-0.3 0.8 -5\n1.6 -0.6 0.0\n"
              "That is, number of points followed by the points.\n"
              "Points are separated by newlines,"
              " co-ordinates are separated by spaces.\n"
              "\"-random\" followed by a space and the number of points will"
              " run the program on that number of random points.\n"
              "\"-debug\" will produce logs of running.\n"
              "One last note: the time taken with stdin input is wrong.\n");
      exit(0);
    }
  }
  if (random_on) {
    num_points = atoi(argv[argc - 1]);
    randomise(num_points);
  }
  else {
    read_points(from_file, argv[argc - 1]);
  }
  //Computing the Delauany triangulation
  ti = clock();
  del = make_list();
  bowyer_watson();
  ti = clock() - ti;
  double time_taken = ((double)ti)/CLOCKS_PER_SEC;
  fprintf(stderr, "Delaunay completed - took extra %lf seconds\n", time_taken);

  voronoi();
  ti = clock() - ti;
  time_taken = ((double)ti)/CLOCKS_PER_SEC;
  fprintf(stderr, "Voronoi completed - took extra %lf seconds\n", time_taken);

  if(debug) {
    print_points();
    print_DLL(del);
    print_voronoi();
  }

  num_tetras = list_length(del);
  max = num_tetras < num_points ? num_points : num_tetras;
  color = calloc(3 * max, sizeof(double));
  //Create the colors
  for (i = 0; i < max; ++ i) {
  #ifdef _WIN32
    color[3*i] = (double)(rand()) / RAND_MAX;
    color[3*i + 1] = (double)(rand()) / RAND_MAX;
    color[3*i + 2] = (double)(rand()) / RAND_MAX;
  #else
    color[3*i] = drand48();
    color[3*i + 1] = drand48();
    color[3*i + 2] = drand48();
  #endif
  }

  //Display here
  bounding_box();
  glutInit (&argc, argv);
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize (480,480);
  glutInitWindowPosition (100,100);
  glutCreateWindow("Del");
  glutDisplayFunc(display);
  glutKeyboardFunc(kbd);
  glutReshapeFunc(reshape);
  init();
  glutMainLoop();

  //Clean up
  empty_dll(del);
  free(p);
  free(color);
  return 0;
}
```

Two Dimensional C Code Snippets

```c
//Same Header files needed

//Global variables
double *p; //p for points - our array of points
int num_points; //Will hold the number of points in consideration
int GRID_SIZE; //Order of hilbert curves;
int BOX_SIZE; //size of viewing box
DLL *del; //Stores Delaunay triangulation
Polygon_2D **voro; //Stores the voronoi diagram

/**************************************************************************
These data structures have vertices as integer labels, which will pull
from an array of points when actual co-ordinates are needed.
*************************************************************************/
```

```c
typedef struct Edge {
  int from, to; //2 vertices as integer labels
} Edge;

typedef struct Triangle {
  int vert_1, vert_2, vert_3; //Three vertices of triangle as integer labels
  Edge edge_1, edge_2, edge_3; //Three edges of the triangle
  //adjacent[i] is pointer to the triangle sharing edge_i+1 - NULL if none exist
  struct Triangle* adjacent[3];
  double circum[2]; //Circumcentre of the triangle
  int checked; //Has the triangle been checked in step j of Bowyer Watson?
} Triangle;

struct Polygon_2D { //An array of vertices, and a boolean unbounded
  double* v;
  int capacity;
  int num_items;
  int unbounded;
};

//Make a ploygon with capacity cap
Polygon_2D *make_polygon(int cap) {
  Polygon_2D *pg = malloc(sizeof(Polygon_2D));
  if (pg == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  pg->capacity = cap;
  pg->v = malloc(sizeof(double) * pg->capacity);
  if (pg->v == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  pg->num_items = 0;
  pg->unbounded = 0;
  return pg;
}

//Add point (a1, a2) to polygon *pg
void add_to_polygon(double a1, double a2, Polygon_2D *pg) {
  if((pg->num_items + 2) > pg->capacity) {
    pg->capacity *= 2;
    pg->v = realloc(pg->v, sizeof(double) * pg->capacity);
    if(pg == NULL) {
      fprintf(stderr, "ERROR: realloc failed\n");
      exit(-1);
    }
  }
  pg->v[pg->num_items++] = a1;
  pg->v[pg->num_items++] = a2;
}

//Calculates the determinant of the 3x3 matrix with rows a,b,c
double determinant(double *a, double *b, double *c) {
  return a[0] * (b[1] * c[2] - b[2] * c[1]) +
         a[1] * (b[2] * c[0] - b[0] * c[2]) +
         a[2] * (b[0] * c[1] - b[1] * c[0]);
}

//Forms a 3x3 matrix using the vertices of the triangle and homogenising 1
//and then computes the determinant of this matrix.
double triangle_determinant(Triangle *t) {
  double a[3];
  double b[3];
  double c[3];
  a[0] = 1;
  a[1] = p[2 * t->vert_1];
  a[2] = p[2 * t->vert_1 + 1];
  b[0] = 1;
  b[1] = p[2 * t->vert_2];
  b[2] = p[2 * t->vert_2 + 1];
  c[0] = 1;
  c[1] = p[2 * t->vert_3];
  c[2] = p[2 * t->vert_3 + 1];
  return determinant(a, b, c);
}

//Creates a postively oriented triangle with the vertices given as parameters
Triangle* make_triangle(int vert_1, int vert_2, int vert_3) {
  Triangle* t = malloc(sizeof(Triangle));
  if (t == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  t->vert_1 = vert_1;
  t->vert_2 = vert_2;
  t->vert_3 = vert_3;
  double check = triangle_determinant(t);
  //check should be greater than 0 is the triangle is counter-clockwise
  //reversing labels reverses the sign of check - and thus orientation
  if (check < 0) {
    t->vert_2 = vert_3;
    t->vert_3 = vert_2;
  }
  else if (check == 0) {
    fprintf (stderr,
        "Error: %d, %d, %d are collinear. Tried to make triangle\n",
        t->vert_1, t->vert_2, t->vert_3);
    exit(-1);
  }
  t->edge_1.from = t->vert_1;
  t->edge_1.to = t->vert_2;
  t->edge_2.from = t->vert_2;
  t->edge_2.to = t->vert_3;
  t->edge_3.from = t->vert_3;
  t->edge_3.to = t->vert_1;

  t->adjacent[0] = t->adjacent[1] = t->adjacent[2] = NULL;
  t->circum[0] = t->circum[1] = 0;
  t->checked = -1;

  return t;
}

/******************************************************************************
In the bowyer_watson algorithm we start by using a big triangle which
surrounds all of the points in consideration.
This big triangle must be removed at the end of the algorithm, so this
function checks if Triangle *t shares a vertex with this big triangle -
the big triangle has vertices n, n + 1 and n + 2
******************************************************************************/
int shares_vertex_bigt(Triangle *t, int n) {
  int b = t->vert_1;
  if ((b == n) || (b == n + 1) || (b == n + 2))
    return 1;

  b = t->vert_2;
  if ((b == n) || (b == n + 1) || (b == n + 2))
    return 1;

  b = t->vert_3;
  if ((b == n) || (b == n + 1) || (b == n + 2))
    return 1;

  return 0;
}

//Checks if e1 == e2 (edges are not directional)
int equal(Edge e1, Edge e2){
  if (e1.to == e2.to && e1.from == e2.from)
    return 1;
  if (e1.to == e2.from && e1.from == e2.to)
    return 1;
  return 0;
}

//The stucture to hold triangle pointers will be a DLL (Doubly Linked List)
//The functions on the DLL are the same as 3D,
//but with triangle pointers instead of tetrahedron pointers.
typedef struct DLL_NODE {
  Triangle *data;
  struct DLL_NODE *next, *prev;
} DLL_NODE;

struct DLL{
  DLL_NODE *first, *last;
};

//The code for a push down Edge stack and Triangle pointer Stack
//Is left out here, same functions as in the 3D case
//Stack is an edge stack
typedef struct Stack{
  int top_index;
  int capacity;
  Edge *item;
} Stack;

typedef struct Triangle_Stack{
  int top_index;
  int capacity;
  Triangle **item;
} Triangle_Stack;

//The code for Hilbert sorting follows
//rotate/flip a quadrant appropriately.
void rot(int quad, int *x, int *y, int w) {
  int temp;
  if (quad == 0) {
    temp = *x;
    *x = *y;
    *y = temp;
  }
  else if (quad == 1) {
    *y = *y - w;
  }
  else if (quad == 2) {
    *x = *x - w;
    *y = *y - w;
  }
  else {
    temp = *x;
    *x = w - *y - 1;
    *y = w * 2 - temp - 1;
  }
}

//Converts a 2D integer point (x,y) to a 1D distance, with grid resolution n
int xy2d (int n, int x, int y) {
  int r;
  int max;
  int w;
  int temp;
  int quad;
  int rx, ry, d = 0;
  if (x >= y) max = x;
  else max = y;
  r = floor(log(max)/log(2)) + 1;
  w = (int)pow(2, r - 1);
  if ((n % 2) != (r % 2)) {
    temp = x;
    x = y;
    y = temp;
  }
  while (r != 0) {
    rx = (x & w) > 0;
    ry = (y & w) > 0;
    quad = (3 * rx) ^ ry;
    d += w * w * quad;
    rot(quad, &x, &y, w);
    r = r - 1;
    w = w/2;
  }
  return d;
}

//Crudely converts a double to an int.
int to_int(double p) {
  return (int) ((p + BOX_SIZE) * 100);
}

//Comparison function to be used to sort the array p
int cmpfunc (const void * a, const void * b) {
  return (xy2d(GRID_SIZE, to_int(p[2 * *(int*)a]),
                          to_int(p[2 * *(int*)a + 1])) -
          xy2d(GRID_SIZE, to_int(p[2 * *(int*)b]),
                          to_int(p[2 * *(int*)b + 1])));
}

//Calls qsort on an array of indexes. These indexes will sort p.
void hilbert_sort(int *indexes) {
  qsort(indexes, num_points, sizeof(int), cmpfunc);
}

//Function to sort the global array p with Hilbert sorting.
void sort() {
  int i;
  int *indexes;
  int check;
  double max = 0;
  double temp;
  double *temp_array;

  indexes = malloc(num_points * sizeof(int));
  if (indexes == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  for (i = 0; i < num_points; ++i) {
    indexes[i] = i;
    temp = fabs(p[2 * i]);
    if(temp > max) max = temp;
    temp = fabs(p[2 * i + 1]);
    if(temp > max) max = temp;
  }

  max = ceil(max);
  BOX_SIZE = (int)max;
  GRID_SIZE = 256;
  check = BOX_SIZE * 2 * 100;
  GRID_SIZE = floor(log(check)/log(2)) + 1;

  hilbert_sort(indexes);
  temp_array = malloc(2 * (num_points + 3) * sizeof(double));
  if (temp_array == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }
  for(i = 0; i < num_points; ++i) {
    temp_array[2 * i] = p[2 * indexes[i]];
    temp_array[2 * i + 1] = p[2 * indexes[i] + 1];
  }
  free(indexes);
  free(p);
  p = temp_array;
}

//square the double d
double sq (double d) {
  return d * d;
}

//Checks if the point (d0,d1) lies inside the positively oriented
//triangle *Tri's circumcircle.
int in_circle(Triangle *Tri, double d0, double d1) {
  double a[3], b[3], c[3];
  a[0] = p[2 * Tri->vert_1] - d0;
  a[1] = p[2 * Tri->vert_1 + 1] - d1;
  a[2] = sq(p[2 * Tri->vert_1] - d0) +
         sq(p[2 * Tri->vert_1 + 1] - d1);

  b[0] = p[2 * Tri->vert_2] - d0;
  b[1] = p[2 * Tri->vert_2 + 1] - d1;
  b[2] = sq(p[2 * Tri->vert_2] - d0) +
         sq(p[2 * Tri->vert_2 + 1] - d1);

  c[0] = p[2 * Tri->vert_3] - d0;
  c[1] = p[2 * Tri->vert_3 + 1] - d1;
  c[2] = sq(p[2 * Tri->vert_3] - d0) +
         sq(p[2 * Tri->vert_3 + 1] - d1);

  if (determinant(a,b,c) <= 0)
    return 0;
  else
    return 1;
}

//Checks if edge e is shared with a triangle in list.
//the search starts at node in list, and moves backwards, then does forwards.
int shared_edge_in_graph(Edge e, DLL_NODE *node, DLL *list) {
  DLL_NODE *temp;

  temp = node->prev;
  while (temp != NULL) {
    if (equal(e, temp->data->edge_1))
      return 1;
    else if (equal(e, temp->data->edge_2))
      return 1;
    else if (equal(e, temp->data->edge_3))
      return 1;
    temp = temp->prev;
  }

  temp = node->next;
  while (temp != NULL) {
    if (equal(e, temp->data->edge_1))
      return 1;
    else if (equal(e, temp->data->edge_2))
      return 1;
    else if (equal(e, temp->data->edge_3))
      return 1;
    temp = temp->next;
  }

  return 0;
}

/*******************************************************************************
Recursively checks the neighbours of Triangle tri. If a neighbour is bad it is
added to DLL bad and checked itself. Otherwise neighbour is marked as checked.
At each step, triangles are ignored that a have a checked value equal to run.
*******************************************************************************/
void check_neighbours(Triangle *tri, int run, DLL *bad) {
  DLL_NODE *node;
  Triangle *nbhr;
  int i;

  tri->checked = run;
  for(i = 0; i < 3; ++i) {
    nbhr = tri->adjacent[i];
    //Check if the neighbour is actually a triangle
    if(nbhr != NULL) {
      //Only check each triangle once
      if(nbhr->checked < run) {
        if (in_circle(nbhr, p[2 * run], p[2 * run + 1])) {
          node = find_node(nbhr, del);
          add_to_list(nbhr, bad);
          remove_node(node, del);
          check_neighbours(nbhr, run, bad);
        }
        else {
          nbhr->checked = run;
        }
      }
    }
  }
}

//Checks if tri and nbhr match on e, and if so updates tri's adjacency
int find_matching_edge(Triangle *tri, Edge e, Triangle* nbhr) {
  if (equal(tri->edge_1, e)) {
    tri->adjacent[0] = nbhr;
    return 1;
  }
  if (equal(tri->edge_2, e)) {
    tri->adjacent[1] = nbhr;
    return 1;
  }
  if (equal(tri->edge_3, e)) {
    tri->adjacent[2] = nbhr;
    return 1;
  }
  fprintf(stderr,"Found no match for an edge in a triangle\n");
  return 0;
}

/*******************************************************************************
```

```c
Checks if edge e is shared by any triangle in a search starting at node and
moving backwards. If it is, the triangle pointed to by node has
adjacent[edge_no] updated with the found triangle.
****************************************************************************/
void find_adjacent_triangle_to_edge(Edge e, DLL_NODE *node, int edge_no) {
  DLL_NODE *comp = node->prev;

  if (node->data->adjacent[edge_no] == NULL) {
    while(comp != NULL) {
        if (equal(e, comp->data->edge_1)) {
          comp->data->adjacent[0] = node->data;
          node->data->adjacent[edge_no] = comp->data;
          break;
        }
        else if (equal(e, comp->data->edge_2)) {
          comp->data->adjacent[1] = node->data;
          node->data->adjacent[edge_no] = comp->data;
          break;
        }
        else if (equal(e, comp->data->edge_3)) {
          comp->data->adjacent[2] = node->data;
          node->data->adjacent[edge_no] = comp->data;
          break;
        }
      comp = comp->prev;
    }
  }
}

//Finds all adjacencies moving backwards in a list starting at start
void find_adjacent(DLL_NODE *start) {
  Edge edge;
  DLL_NODE *node;
  node = start;
  while (node != NULL) {
    //check edge 1
    edge = node->data->edge_1;
    find_adjacent_triangle_to_edge(edge, node, 0);
    //check edge 2
    edge = node->data->edge_2;
    find_adjacent_triangle_to_edge(edge, node, 1);
    //check edge 3
    edge = node->data->edge_3;
    find_adjacent_triangle_to_edge(edge, node, 2);
    node = node->prev;
  }
}

//Make all triangles which tri points to, point to NULL instead of back to tri
void delete_ties(Triangle *tri) {
  int i, j;
  for(i = 0; i < 3; ++i) {
    if(tri->adjacent[i] != NULL) {
      for(j = 0; j < 3; ++j) {
        if(tri->adjacent[i]->adjacent[j] == tri) {
          tri->adjacent[i]->adjacent[j] = NULL;
          break;
        }
      }
    }
  }
}

//This algorithm produces the Delaunay triangulation of our points
void bowyer_watson() {
  int i;
  Edge edge;
  int shared_edge;
  int found_bad;
  Triangle *tri_1;
  Triangle *tri_2;
  DLL *bad = make_list();
  DLL_NODE *node;
  DLL_NODE *temp;
  Stack *polygon = make_stack();
  Triangle_Stack *border_triangles = make_tri_stack();

  //Add big triangle vertices to array of points
  p[2 * num_points] = -100000;
  p[2 * num_points + 1] = -100000;
  p[2 * num_points + 2] = 100000;
  p[2 * num_points + 3] = -100000;
  p[2 * num_points + 4] = 0;
  p[2 * num_points + 5] = 100000;

  Triangle *big_tri = make_triangle(num_points, num_points + 1, num_points + 2);
  add_to_list (big_tri, del);
  for (i = 0; i < num_points; ++i) {
    empty_dll(bad);
    node = del->first;
    found_bad = 0;
    while(!found_bad) { //Find one bad triangle
      if (node == NULL) {
        fprintf(stderr,"ERROR: Arithmetic error - found no bad triangle\n");
        exit(-1);
      }
      if (in_circle(node->data, p[2 * i], p[2 * i + 1])) {
        add_to_list(node->data, bad);
        remove_node(node, del);
        found_bad = 1;
      }
      else {
        node = node->next;
      }
    }
    check_neighbours(bad->first->data, i, bad); //Find all bad triangles
    node = bad->first;
    while(node != NULL) {
      //check edge 1
      edge = node->data->edge_1;
      shared_edge = shared_edge_in_graph(edge, node, bad);
      if(!shared_edge) {
        push(edge, polygon);
        push_tri(node->data->adjacent[0], border_triangles);
      }
      //check edge 2
      edge = node->data->edge_2;
      shared_edge = shared_edge_in_graph(edge, node, bad);
      if(!shared_edge) {
        push(edge, polygon);
        push_tri(node->data->adjacent[1], border_triangles);
      }
      //check edge 3
      edge = node->data->edge_3;
      shared_edge = shared_edge_in_graph(edge, node, bad);
      if(!shared_edge) {
        push(edge, polygon);
        push_tri(node->data->adjacent[2], border_triangles);
      }
      node = node->next;
    }
    //Find the Delaunay cavity of the input point.
    //The first run of our while loop is slighty different, so is separate
    edge = top(polygon);
    add_to_list(make_triangle(edge.from, edge.to, i), del);
    node = del->first; //The different line - marks where new triangles start
    tri_2 = top_tri(border_triangles);
    if (tri_2 != NULL) {
      tri_1 = del->first->data;
      find_matching_edge(tri_1, edge, tri_2);
      find_matching_edge(tri_2, edge, tri_1);
    }
    pop_tri(border_triangles);
    pop(polygon);

    while (!is_empty_stack (polygon)) {
      edge = top(polygon);
      add_to_list(make_triangle(edge.from, edge.to, i), del);
      tri_2 = top_tri(border_triangles);
      if (tri_2 != NULL) {
        tri_1 = del->first->data;
        find_matching_edge(tri_1, edge, tri_2);
        find_matching_edge(tri_2, edge, tri_1);
      }
      pop_tri(border_triangles);
      pop(polygon);
    }
    //Fill in adjacencies for newly added triangles.
    find_adjacent(node);
  }
  node = del->first;
  while (node != NULL){ //Remove ties with the super triangle
    if (shares_vertex_bigt(node->data, num_points) == 1) {
      temp = node;
      node = node->next;
      delete_ties(temp->data);
      free(temp->data);
      remove_node(temp, del);
    }
    else {
      node = node->next;
    }
  }
  //Clean up
  empty_dll(bad);
  free(bad);
  free_stack(polygon);
  free_tri_stack(border_triangles);
}

//Finds the circumcentre of triangle pointed to by t and stores it in result
void circumcentre(double *result, Triangle *t) {
  double slope_1, slope_2, mid_x1, mid_x2, mid_y1, mid_y2;
  double x1 = p[2 * t->vert_1];
  double x2 = p[2 * t->vert_2];
  double x3 = p[2 * t->vert_3];
  double y1 = p[2 * t->vert_1 + 1];
  double y2 = p[2 * t->vert_2 + 1];
  double y3 = p[2 * t->vert_3 + 1];
  double temp;

  //slight complication for slope 0 lines:
  if(y1 == y2) {
    temp = y2;
    y2 = y3;
    y3 = temp;
    temp = x2;
    x2 = x3;
    x3 = temp;
  }
  else if(y3 == y2) {
    temp = y2;
    y2 = y1;
    y1 = temp;
    temp = x2;
    x2 = x1;
    x1 = temp;
  }

  mid_x1 = (x1 + x2) / 2;
  mid_x2 = (x2 + x3) / 2;
  mid_y1 = (y1 + y2) / 2;
  mid_y2 = (y2 + y3) / 2;

  slope_1 = (x2 - x1) / (y1 - y2);
  slope_2 = (x2 - x3) / (y3 - y2);

  result[0] = ((mid_y1 - slope_1 * mid_x1) - (mid_y2 - slope_2 * mid_x2)) /
              (slope_2 - slope_1);
  result[1] = (slope_1 * result[0]) + mid_y1 - (slope_1 * mid_x1);
}

//finds a point on the line ab in the direction ab and stores in r.
void point_on_line(double a1, double a2, double b1, double b2, double *r) {
  double min;
  double d, e;
  d = (fabs(b1 - a1) == 0) ? 10000 : fabs(b1 - a1);
  e = (fabs(b2 - a2) == 0) ? 10000 : fabs(b2 - a2);
  min = d < e ? d : e;
  r[0] = (b1 - a1) * (25  / min) + a1;
  r[1] = (b2 - a2) * (25  / min) + a2;
}

//Stores the midpoint of Edge in e in array midpoint
void midpoint(double *midpoint, Edge e) {
  midpoint[0] = (p[2 * e.from] + p[2 * e.to]) / 2;
  midpoint[1] = (p[2 * e.from + 1] + p[2 * e.to + 1]) / 2;
}

//Returns 1 if points (a_1,a_2), (b_1,b_2), (c_1,c_2) counter-clockwise, else 0
int is_counterclock(double a_1, double a_2,
                    double b_1, double b_2,
                    double c_1, double c_2) {
  double a[3], b[3], c[3];
  a[0] = 1; a[1] = a_1; a[2] = a_2;
  b[0] = 1; b[1] = b_1; b[2] = b_2;
  c[0] = 1; c[1] = c_1; c[2] = c_2;
  if (determinant(a,b,c) > 0) return 1;
  else return 0;
}
/***************************************************************************
Finds a point on the perpendicular bisector of edge e,
belonging to a triangle with circumcentre cc, and stores the result in temp.
We guarantee that temp will form a clockwise triangle with e.
***************************************************************************/
void point_on_bisector(Edge e, double *temp, double *cc) {
  double slope;
    /*************************************************************************
    Three cases, in order:
    1. Circumcentre lies on vertical edge of triangle
    2. Circumcentre lies on horizontal edge of triangle
    3. Circumcentre lies on an angled edge of triangle
    *************************************************************************/
  if (p[2 * e.from] == p[2 * e.to]) {
    temp[0] = cc[0] + 2;
    temp[1] = cc[1];
  }
  else if (p[2 * e.from + 1] == p[2 * e.to + 1]) {
    temp[0] = cc[0];
    temp[1] = cc[1] + 2;
  }
  else {
    slope = (p[2 * e.from] - p[2 * e.to]) /
            (p[2 * e.to + 1] - p[2 * e.from + 1]);
    temp[0] = cc[0] + 2;
    //y = y1 + m(x - x1)
    temp[1] = cc[1] +
                slope * (temp[0] - cc[0]);
  }
  //Makes sure the point lies on the correct side of edge e
  if (is_counterclock(p[2 * e.from], p[2 * e.from + 1],
                      p[2 * e.to], p[2 * e.to + 1],
                      temp[0],        temp[1])) {
    point_on_line(temp[0], temp[1], cc[0], cc[1], temp);
  }
}

//Finds an unbounded vertice corresponding to an unbounded edge
//formed by edge e in Triangle t, storing in r.
void compute_unbounded_vertice(Edge e, Triangle t, Polygon_2D *pg) {
  double temp[2];
  double cc[2];

  midpoint(temp, e);
  cc[0] = t.circum[0];
  cc[1] = t.circum[1];
  /**************************************************************************
  Three cases, in order:
  1. circumcentre lies on the edge e of triangle.
  2. circumcentre outside of triangle.
  3. circumcentre inside triangle.
  ****************************************************************************/
  if ((fabs(temp[0] - cc[0]) <  1e-7) && (fabs(temp[1] - cc[1]) < 1e-7)) {
      point_on_bisector(e, temp, cc);
  }
  else if (is_counterclock (p[2 * e.from], p[2 * e.from + 1],
                            p[2 * e.to], p[2 * e.to + 1],
                            cc[0],   cc[1])) {
    //reflect circumcentre about midpoint
    point_on_line(cc[0], cc[1], temp[0], temp[1], temp);
  }
  else {
    //reflect midpoint about circumcentre
    point_on_line(temp[0], temp[1], cc[0], cc[1], temp);
  }
  add_to_polygon(temp[0], temp[1], pg);
}

//finds the edges in *t that have v as a vertice, storing in r
void edges_with_vert(int v, Triangle *t, int *r) {
  int count = 0;
  if((t->edge_1.from == v) || (t->edge_1.to == v)) r[count++] = 0;
  if((t->edge_2.from == v) || (t->edge_2.to == v)) r[count++] = 1;
  if((t->edge_3.from == v) || (t->edge_3.to == v)) r[count++] = 2;
}

//Copies pg1 into pg2, in reverse order of points.
void copy_pg_reverse(Polygon_2D *pg1, Polygon_2D *pg2) {
  int i;
  for(i = ((pg1->num_items / 2) - 1); i >= 0; --i) {
    add_to_polygon(pg1->v[2 * i], pg1->v[2 * i + 1], pg2);
  }
}

//Copies pg1 into pg2
void copy_pg(Polygon_2D *pg1, Polygon_2D *pg2) {
  int i;
  for(i = 0; i < (pg1->num_items/ 2); ++i) {
    add_to_polygon(pg1->v[2 * i], pg1->v[2 * i + 1], pg2);
  }
}

//Computes the Voronoi region for v a vertice of triangle *start, storing in *pg
void voronoi_face(Triangle *start, int v, Polygon_2D *pg, Polygon_2D *temp_pg) {
  Triangle *current, *prev, *next;
  //Will hold the indexes of the edges that share a particular vetice
  int edges[2], start_edges[2];
  int i = 0;
  int done = 0;
  Edge e;

  if(start == NULL) {
    fprintf(stderr, "ERROR: started with NULL triangle\n");
    exit(-1);
  }
  temp_pg->num_items = 0;
  add_to_polygon(start->circum[0], start->circum[1], temp_pg);
  edges_with_vert(v, start, start_edges);
  while((i < 2) && !done) {//i == 0 first direction, i == 1 second direction
    prev = start;
    if(start->adjacent[start_edges[i]] != NULL) {
      next = start->adjacent[start_edges[i]];
      while(next != start) {//Not returned to starting point
        current = next;
        if(i == 1) add_to_polygon(current->circum[0], current->circum[1], pg);
        else add_to_polygon(current->circum[0], current->circum[1], temp_pg);
        edges_with_vert(v, current, edges);
        if(current->adjacent[edges[0]] != prev) {
          if(current->adjacent[edges[0]] == NULL) {
            if(edges[0] == 0) e = current->edge_1;
            else if(edges[0] == 1) e = current->edge_2;
            else e = current->edge_3;
            if(i == 1) compute_unbounded_vertice(e, *current, pg);
            else {
              compute_unbounded_vertice(e, *current, temp_pg);
              copy_pg_reverse(temp_pg, pg);
            }
            pg->unbounded = 1;
            break;
          }
          next = current->adjacent[edges[0]];
          prev = current;
        }
        else if(current->adjacent[edges[1]] == prev) {
          fprintf(stderr, "ERROR: incorrect adjacency\n");
          exit(-1);
        }
        else {
          if(current->adjacent[edges[1]] == NULL) {
            if(edges[1] == 0) e = current->edge_1;
            else if(edges[1] == 1) e = current->edge_2;
            else e = current->edge_3;
            if(i == 1) compute_unbounded_vertice(e, *current, pg);
            else {
              compute_unbounded_vertice(e, *current, temp_pg);
              copy_pg_reverse(temp_pg, pg);
            }
            pg->unbounded = 1;
            break;
          }
          next = current->adjacent[edges[1]];
          prev = current;
        }
      }
      if(next == start) done = 1;
    }
    else {
      if(start_edges[i] == 0) e = start->edge_1;
      else if(start_edges[i] == 1) e = start->edge_2;
      else e = start->edge_3;
      if(i == 1) {
        compute_unbounded_vertice(e, *start, pg);
      }
      else {
        compute_unbounded_vertice(e, *start, temp_pg);
        copy_pg_reverse(temp_pg, pg);
      }
      pg->unbounded = 1;
    }
    ++i;
  }
  if(done) copy_pg(temp_pg, pg);
}

//Produces the voronoi diagram for our points.
void voronoi() {
  Triangle *t;
  DLL_NODE *node;
  int count = 0;
  double temp[2];
  //Will hold if a vertice is done.
  int *done = calloc(num_points, sizeof(int));
  Polygon_2D *temp_pg = make_polygon(3000);

  voro = malloc(num_points * sizeof(Polygon_2D*));
  if (voro == NULL) {
    fprintf(stderr, "ERROR: malloc failed\n");
    exit(-1);
  }

  node = del->first;
  while (node != NULL) {
    circumcentre(temp, node->data);
    node->data->circum[0] = temp[0];
    node->data->circum[1] = temp[1];
    node = node->next;
  }

  node = del->first;
  while(node != NULL) {
    t = node->data;
    if(!done[t->vert_1]) {
      voro[count] = make_polygon(30);
      voronoi_face(t, t->vert_1, voro[count], temp_pg);
      done[t->vert_1] = 1;
      ++count;
    }
    if(!done[t->vert_2]) {
      voro[count] = make_polygon(30);
      voronoi_face(t, t->vert_2, voro[count], temp_pg);
      done[t->vert_2] = 1;
      ++count;
    }
    if(!done[t->vert_3]) {
      voro[count] = make_polygon(30);
      voronoi_face(t, t->vert_3, voro[count], temp_pg);
      done[t->vert_3] = 1;
      ++count;
    }
    if(count == num_points) break;
    node = node->next;
  }
  free(temp_pg);
}

//Drawing routines left out here, similar to 3D case.

//read_points and randomise left out here, just assume that p holds input
//Point set and num_p holds the number of points in p.

//Does memory allocation of global arrays and calls fucntions
int main() {
  //Hilbert sorting of the array p
  sort();

  //Computing the Delauany triangulation
  del = make_list();
  bowyer_watson();

  //Computing the Voronoi diagram
  voronoi();
  return 0;
}
```