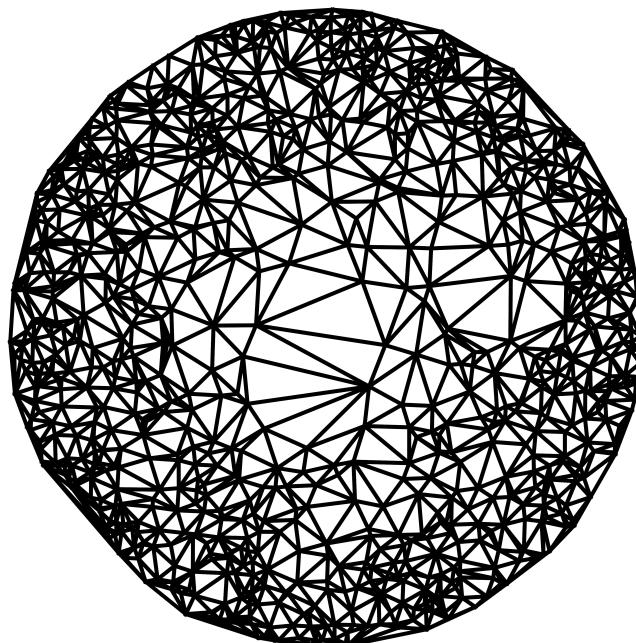




**Trinity College Dublin**  
Coláiste na Tríonóide, Baile Átha Cliath  
**The University of Dublin**

## Delaunay Triangulations on the GPU

In partial fulfillment of MSc in High-performance Computing in the  
School of Mathematics



Name: Patryk Drozd  
Student ID: 24333177  
Supervision: Jose Refojo  
Date of Submission: 25/09/2025  
Project Code: [https://github.com/drozd324/GPU\\_Delaunay\\_Triangulation](https://github.com/drozd324/GPU_Delaunay_Triangulation)



**Trinity College Dublin**  
Coláiste na Tríonóide, Baile Átha Cliath  
**The University of Dublin**

## **Declaration concerning plagiarism**

I hereby declare that this thesis is my own work where appropriate citations have been given and that it has not been submitted as an exercise for a degree at this or any other university.

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>.

Name: Patryk Drozd

Student ID: 24333177

Signature:

A handwritten signature in black ink, appearing to read "Patryk Drozd".

Date: 25/09/2025

## **Acknowledgements**

Write acknowledgements to your supervisor, classmates, friends, family, partner... anyone who supported you during the MSc. Lorem ipsum.

# Contents

Acknowledgements .....	3
1. Delaunay triangulations .....	6
2. The GPU .....	9
3. Algorithms .....	11
3.1. Serial .....	11
3.1.1. Point insertion .....	12
3.1.2. Flipping .....	13
3.1.3. Implementation .....	14
3.1.4. Analysis .....	15
3.2. Parallel .....	16
3.2.1. Insertion .....	18
3.2.2. Flipping .....	19
3.2.3. Implementation .....	20
3.2.4. Analysis .....	20
3.3. Data Structures .....	27
3.4. Profiling Analysis .....	28
3.5. Comment on the Floating point arithmetic and how the maths is funny sometimes .....	28
3.6. Strong and weak scaling on GPUs? .....	28
Bibliography .....	29

## Abstract

Triangulations of a set of points are a very useful mathematical construction to describe properties of discretised physical systems, such as modelling terrains, cars and wind turbines which are commonly used for simulations such as computational fluid dynamics or other physical properties, and even have use in video games for rendering and visualising complex geometries. To paint a picture you may think of a triangulation given a set of points  $P$  to be a bunch of line segments connecting each point in  $P$  in a way such that the edges are non intersecting. A particularly interesting subset of triangulations are Delaunay triangulations (DT). The Delaunay triangulation is a triangulation which maximises all angles in each triangle of the triangulation. Mathematically this gives us an interesting optimization problem which leads to some rich mathematical properties, at least in 2 dimensions, and for the applied size we have a good way to discretize space for the case of simulations with the aid of methods such as Finite Element and Finite Volume methods. Delaunay triangulations in particular are a good candidate for these numerical methods as they provide us with fat triangles, as opposed to skinny triangles, which can serve as good elements in the Finite Element method as they tend to improve accuracy [1].

There are many algorithms which compute Delaunay triangulations (cite some overview paper), however a lot of them use the operation of ‘flipping’ or originally called an ‘exchange’ [2]. This is a fundamental property of moving through triangulations of a set of points to with the goal of obtaining the optimal Delaunay triangulation. This flipping operation involves a configuration of two triangles sharing an edge, forming a quadrilateral with its boundary. The shared edge between these two triangles will be swapped or flipped from the two points at its end to the other two points on the quadrilateral. The original algorithm motivated by ([2]) is hinted to be us this flipping operation to iterate through different triangulations and eventually arrive at the Delaunay triangulation which we desire.

With the flipping operation being at the core of the algorithm, we can notice that it has the possibility of being parallelized. This is desirable as problems which commonly use the DT are run with large datasets and can benefit from the highly parallelisable nature of this algorithm. If we wish to parallelize this idea, and start with some initial triangulation, conflicts would only occur if we chose to flip a configuration of triangles which share a triangle. With some care, this is an avoidable situation leads to a highly scalable algorithm. In our case the hardware of choice will be the GPU which is designed with the SIMD model which is particularly well suited for this algorithm as we are mostly performing the same operations in each iteration of the algorithm in parallel.

The goal of this project was to explore the Delaunay triangulations through both serial and parallel algorithms with the goal of presenting a easy to understand, sufficiently complex parallel algorithm designed with Nvidia’s CUDA programming model for running software on their GPUs.

## 1. Delaunay triangulations

In this section I aim to introduce triangulations and Delaunay triangulations from a mathematical perspective with the foresight to help present the motivation and inspiration for the key algorithms used in this project. For the entirety of this project we only focus on 2 dimensional Delaunay triangulations.

In order to introduce the Delaunay Traingulation we first must define what we mean by a triangulation. In order to create a triangulation we need a set of points which will make up the vertices of the triangles. But first we want to clarify a possible ambiguity about edges.

**Definition 1.1:** For a point set  $P$ , the term edge is used to indicate any segment that includes precisely two points of  $S$  at its endpoints. [3].

Alternatively we could say an edge doesn't contain its endpoints which could be more useful in different contexts. But now we define the triangulation.

**Definition 1.2:** A *triangulation* of a planar point set  $P$  is a subdivision of the plane determined by a maximal set of noncrossing edges whose vertex set is  $P$  [3].

This is a somewhat technical but precise definition. The most important point in Definition 1.2 is that it is a *maximal* set of noncrossing edges which for us means that we will not have any other shapes than triangles in this structure.

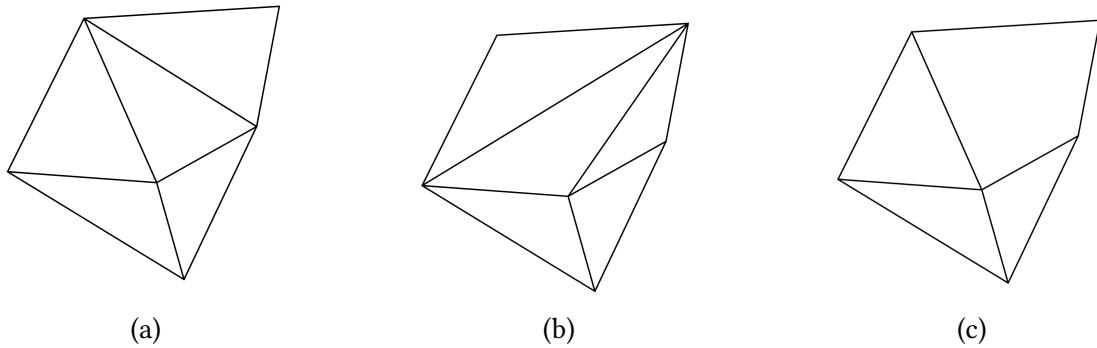


Figure 1: Examples of two triangulations (a) (b) on the same set of points. In (c) an illustration of a non maximal set of edges.

A useful fact about triangulations is that we can know how many triangles our triangulation will contain if given a set of points and its convex hull. For our purposes the convex hull will always be a set which will cover a set of points, in our case the points in our triangulation. This will be useful when we will be storing triangles as we will always know the number of triangles that will be created.

**Theorem 1.1:** Let  $P$  be a set of  $n$  points in the plane, not all collinear, and let  $k$  denote the number of points in  $P$  that lie on the boundary of the convex hull of  $P$ . Then any triangulation of  $P$  has  $2n - 2 - k$  triangles and  $3n - 3 - k$  edges. [4]

A key feature of all of the Delaunay triangulation theorems we will be considering that no three points from the set of points  $P$  which will make up our triangulation will lie on a line and also that no 4 points like on a circle. Motivation for this definition will become more apparent in Theorem 1.2 and following. Definition 1.3 lets us imagine that our points are distributed randomly enough so that our algorithms will work with no degeneracies appearing. This leads us to the following definition.

**Definition 1.3:** A set of points  $P$  is in *general position* if no 3 points in  $P$  are colinear and that no 4 points are cocircular.

From this point onwards we will always assume that the point set  $P$  from which we obtain our triangulation will be in *general position*. This is necessary for the definitions and theorems we will define.

In order to define a Delaunay triangulation we would like to establish the motivation for the definition with another, preliminary definition. A Delaunay triangulation is a type of triangulation which in a sense maximizes smallest angles in a triangulation  $T$ . This idea is formalized by defining an *angle sequence*  $(\alpha_1, \alpha_2, \dots, \alpha_{3n})$  of  $T$  which is an ordered list of all angles of  $T$  sorted from the smallest to largest. With angle sequences we can now compare two triangulations to each other. We can say for two triangulations  $T_1$  and  $T_2$  we write  $T_1 > T_2$  ( $T_1$  is fatter than  $T_2$ ) if the angle sequence of  $T_1$  is lexicographically greater than  $T_2$ . Now we can compare triangulations. And by defining Definition 1.4 are able to define a *Delaunay triangulation*.

**Definition 1.4:** Let  $e$  be an edge of a triangulation  $T_1$ , and let  $Q$  be the quadrilateral in  $T_1$  formed by the two triangles having  $e$  as their common edge. If  $Q$  is convex, let  $T_2$  be the triangulation after flipping edge  $e$  in  $T_1$ . We say  $e$  is a *legal edge* if  $T_1 \geq T_2$  and  $e$  is an *illegal edge* if  $T_1 < T_2$  [3]

**Definition 1.5:** For a point set  $P$ , a *Delaunay triangulation* of  $P$  is a triangulation that only has legal edges. [3]

As per Definition 1.5, Delaunay triangulations wish to only contain legal edges and this provides us with a “nice” triangulation with fat triangles.

**Theorem 1.2 (Empty Circle Property):** Let  $P$  be a point set in general position. A triangulation  $T$  is a Delaunay triangulation if and only if no point from  $P$  is in the interior of any circumcircle of a triangle of  $T$ . [3]

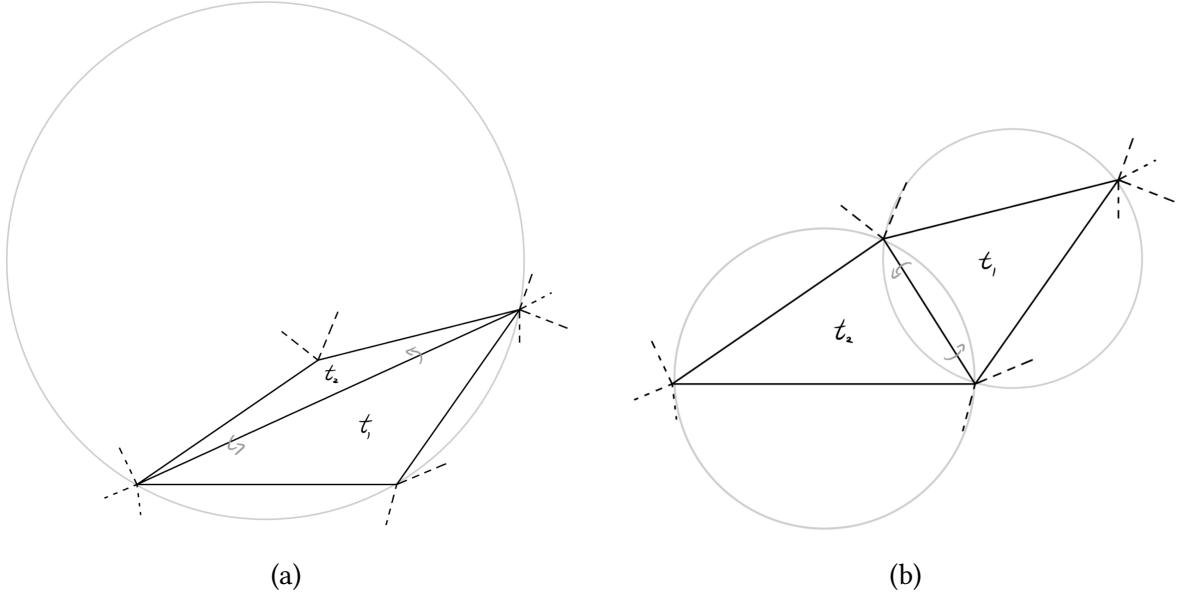


Figure 2: Demonstration of the flipping operation. In (a) A configuration that needs to be flipped illustrated by the circumcircle of  $t_1$  containing the auxillary point of  $t_2$  in its interior. In (b) configuration (a) which has been flipped and no longer needs to be flipped as illustrated by the both circumcircles of  $t_1$  and  $t_2$ .

Theorem 1.2 is the key ingredient in the the Delaunay triangulation algorithms we are going to use. This is because instead of having to compare angles, as would be demanded by Definition 1.5, we are allowed to only perform a computation, involving finding a circumcircle and performing one comparison which would involve determining whether the point not shared by triangles circumcircle is contained inside the circumcircle or not. Algorithms such as initially introduced by Lawson [5] exist which do focus on angle comparisons but are not preferred as they do not introduce desired locality and are more complex.

And finally we present the theorem which guarantees that we will eventually arrive at our desired Delaunay triangulation by stating that we can travel across all possible triangulations of our point set  $P$  by using the fliping operation.

**Theorem 1.3 (Lawson):** Given any two triangulations of a set of points  $P$ ,  $T_1$  and  $T_2$ , there exist a finite sequence of exchanges (flips) by which  $T_1$  can be transformed to  $T_2$ . [2]

## 2. The GPU

The Graphical Processing Unit (GPU) is a type of hardware accelerator originally used to significantly improve running video rendering tasks for example in video games through visualizing the two or three dimensional environments the player would be interacting with or rendering videos in movies after the addition of visual effects. Many different hardware accelerators have been tried and tested for more general use, like Intels Xeon Phis, however the more purpose oriented GPU has prevailed in the market and in performance mainly lead by Nvidia in previous years. Today, the GPU has gained a more general purpose status with the rise of General Purpose GPU (GPGPU) programming as more and more people have noticed that GPUs are very useful as a general hardware accelerator.

The traditional CPU (based on the Von Neumann architecture) which is built to perform *serial* tasks, the CPU is built to be a general purpose hardware for performing all tasks a user would demand from the computer. In contrast the GPU can't run alone and must be used in conjunction to the CPU. The CPU sends compute instructions for the GPU to perform and data is commonly passed between the CPU and GPU.

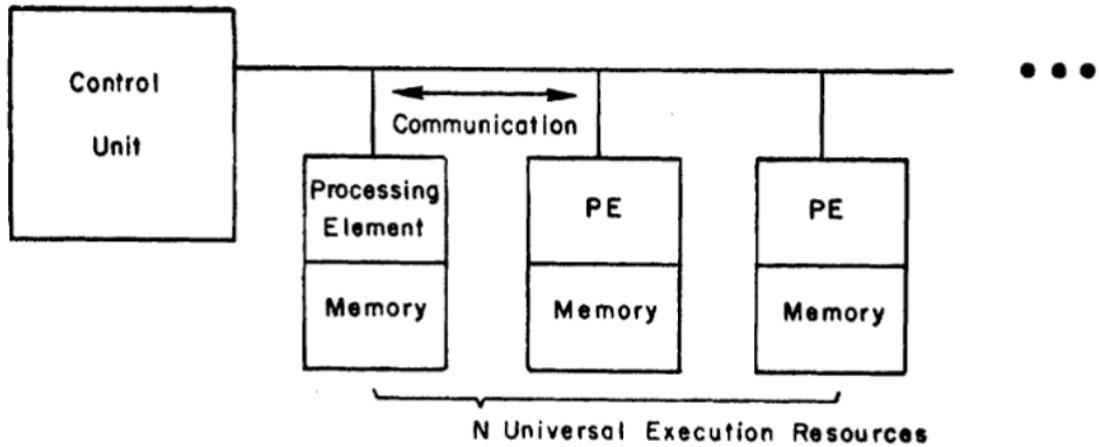


Figure 3: The *Single Instruction Multiple Threads (SIMT)* classification, originally known as an *Array Processor* as illustrated by Michael J. Flynn [6]. The control unit communicates instructions to the  $N$  processing element with each processing unit having its own memory.

What makes the GPU incredibly useful in certain usecases (like the one of this report) is its architecture which is built to enable massively parallelisable tasks. In Flynn's Taxonomy [6], the GPUs architecture is based a subcategory of the Single Instruction Multiple Data (SIMD) classification known as Single Instruction Multiple Threads (SIMT) also known as an Array Processor. The SIMD classification allows for many processing units to perform the same tasks on a shared dataset with the SIMT classification additionally allowing for each processing unit having its own memory allowing for more diverse processing of data.

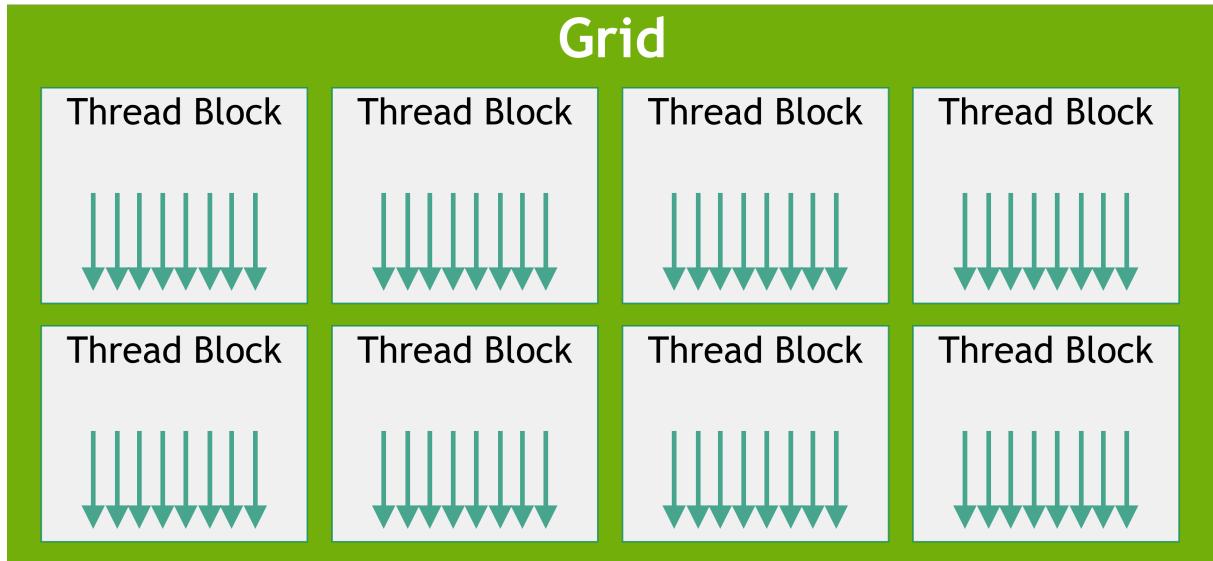


Figure 4: An illustration of the structure of the GPU programming model. As the lowest compute instruction we have a thread block consisting of a number threads  $\leq 1024$ . The thread blocks are contained in a grid. [7]

Nvidia's GPUs take the SIMD model and further develop it. There are three core abstractions which allow Nvidia's GPU model to be successful; a hierarchy of thread groups, shared memories and synchronization [7]. The threads, which represent the theoretical processes which encode programmed instructions, are launched together in groups of 32 known as *warps*. This is the smallest unit of instructions that is executed on the GPU. The threads are further grouped into *thread blocks* which are used as a way of organizing the shared memory to be used by each thread in this thread block. And once more the *thread blocks* grouped into a *grid*.

### 3. Algorithms

In this section we focus on two types of algorithms, serial and parallel, but with a major focus on the parallel algorithm. Commonly algorithms are first developed with a serialized version and only later optimized into parallelized versions if possible. This is how I will be presenting my chosen Delanay Triangulation (DT) algorithms in order to portray a chronological development of ideas used in all algorithms. And so we first begin by explaining the chosen serial version of the DT algorithm.

#### 3.1. Serial

The simplest type of DT algorithm can be stated as follows in Algorithm 1

---

**Algorithm 1:** Lawson Flip algorithm

---

Let  $P$  be a point set in general position. Initialize  $T$  as any triangulation of  $P$ . If  $T$  has an illegal edge, flip the edge and make it legal. Continue flipping illegal edges, moving through the flip graph of  $P$  in any order, until no more illegal edges remain. [3]

---

This algorithm presents with a bit of ambiguity however I believe its a good algorithm to keep in mind when progressing to more complex algorithms as it presents the most important feature in a DT algorithm, that is, checking if an edge in the triangulation is legal, and if its not, we flip it. Most DT algorithms take this core concept and build a more optimized versions of it with as Algorithm 1 has a complexity of  $O(n^2)$  [8].

The next best serial algorithm commonly presented by popular textbooks [4], [9] is the *randomized incremental point insertion* Algorithm 2. When implemented properly this algorithm should have a complexity of  $O(n \log(n))$  [4]. This algorithm is favoured for its relative low complexity and ease of implementation. The construction this algorithm is a bit mathematically involved however the motivation behind the construction of the algorithm is to perform point insertions, and after each point insertion we perform necessary flips to transform the current triangulation into a DT. This in turn reduces the number of flips we need to perform and this is reflected in the runtime complexity.

---

**Algorithm 2:** Randomized incremental point insertion

---

Data: point set  $P$   
Out: Delaunay triangulation  $T$

```
1 | Initialize  $T$  with a triangle enclosing all points in  $P$ 
2 | Compute a random permutation of  $P$ 
3 | for  $p \in P$ 
4 |   Insert  $p$  into the triangle  $t$  containing it
5 |   for each edge  $e \in t$ 
6 |     LegalizeEdge( $e, t$ )
7 |   return  $T$ 
```

---

A significant part of this algorithm is the `FlipEdge` function in Algorithm 3. This function performs the necessary flips, both number of and on the correct edges, for the triangulation in the current iteration of point insertion to become a DT.

---

**Algorithm 3:** LegalizeEdge

---

Data: egde  $e$ , triangle  $t_a$

- 1    **if**  $e$  is illegal
- 2    *flip* with triangle  $t_b$  across edge  $e$
- 3    let  $e_1, e_2$  be the outward facing egdes of  $t_b$
- 4    LegalizeEdge( $e_1, t_b$ )
- 5    LegalizeEdge( $e_2, t_b$ )

---

The contrustions neccesary to explain wyh the *LegalizeEdge* routine created a DT is again slightly mathematically involved but is discussed in [4]. In the following sections we will discuss the point insertion and flipping steps in more detail.

### 3.1.1. Point insertion

Point instertion procedure goes as follows. An initial triangulation is neccesary to begin advance in the point insertion procedure. This is commonly done by adding 3 extra points to our triangulation from which we will construct a *supertriangle* which will contain all of the point in the set we wish to construct the DT. These extra 3 points will later be removed. In our triangulation if there is a point not yet insterted we choose to use it to split the existing triangle in which this point lies in into 3 new triangles. This process is repreated untill no more points are left to insert. The point insertion step would be followed by the *LegaiseEdge* procedure. Figure 5 illustrates this process.

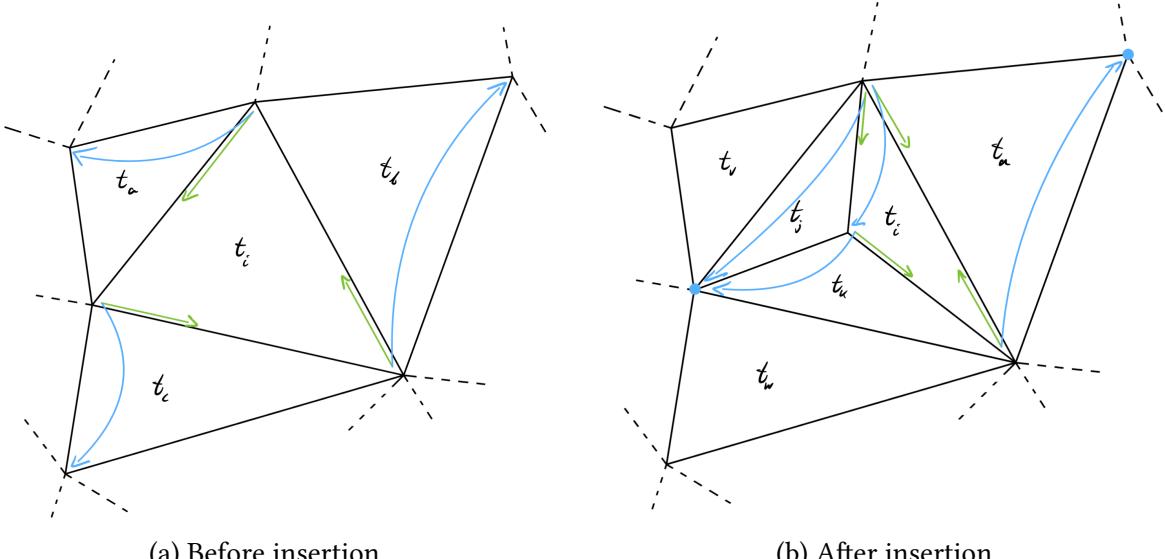


Figure 5: An illustration of the point insertion in step 4 of Algorithm 2. In figure (a) the center most triangle  $t_i$  will be then triangle in which a point will be chosen for insertion. Triangle  $t_i$  knows its neighbours across each edge represented by the green arrows and knows the points opposite each of these edges. After the point it inserted (b),  $t_i$  is moved and two new triangles  $t_j, t_k$  are created to accomodate the new point. Each new trianlge  $t_i, t_j, t_k$  can be fully constructed from the previously existing  $t_i$  and each neighbour of  $t_i$  in (a) has its neighbours updated to reflect the insertion. The neighbouring triangles opposite points are updated by accesing the opposite point across the edge of the neighbouring triangle and obtaining the index of the edge which has the triangle currently being split. The index of the opposite point will allways be 0 by construction. The neighbouring triangle is also updated similarly but with the appropriate index which will be the one of the triangle who's modifying the neighbouring triangle.

It might be nice to see results from just running the point insertion algorithm by itself, without the flipping which would take place in between which will be further explored in the next section. In Figure 6 we see the result after the super triangle points and their corresponding triangles have been removed. It is good to note that the point insertion algorithm is in general a triangulation algorithm.

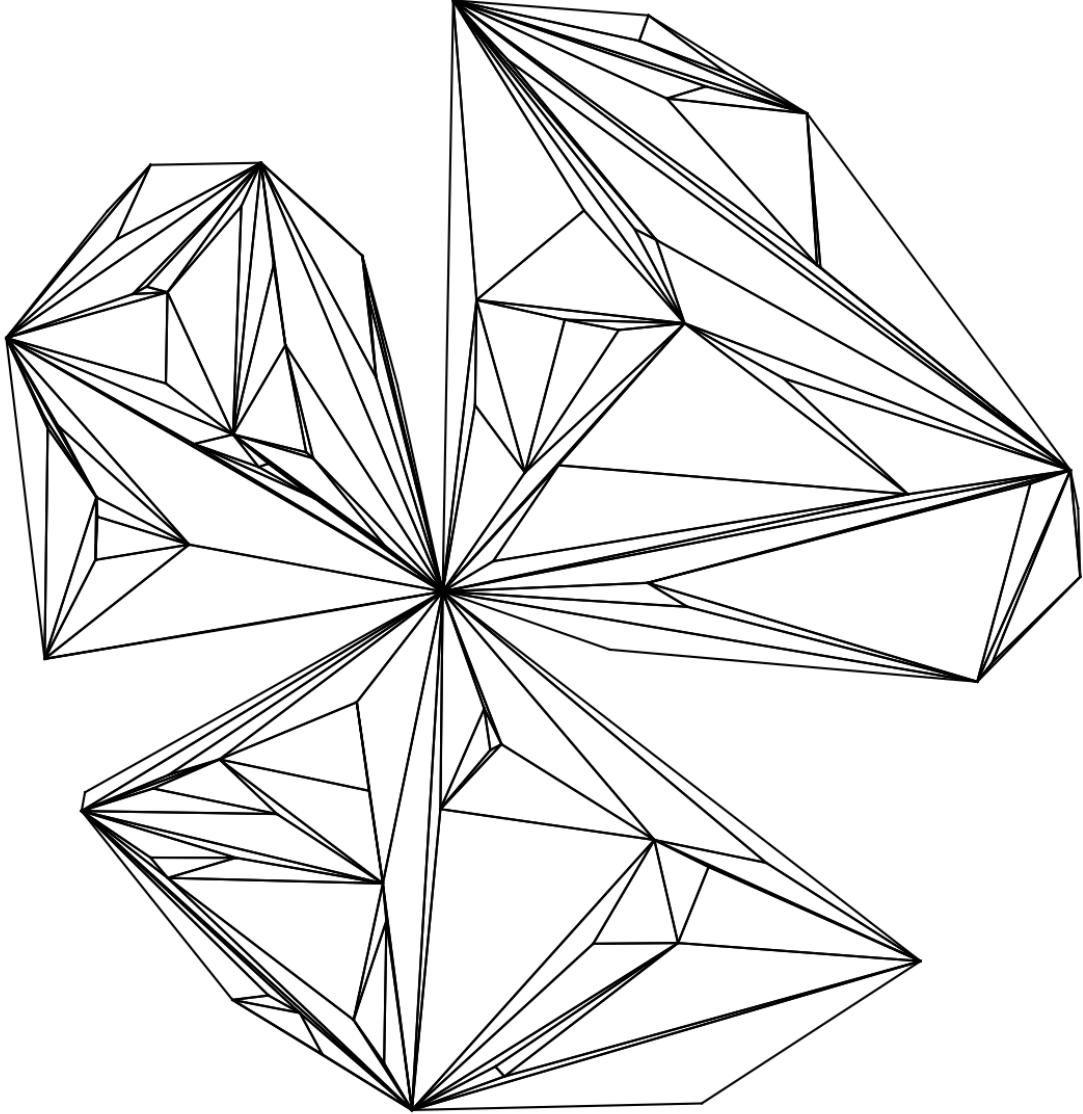
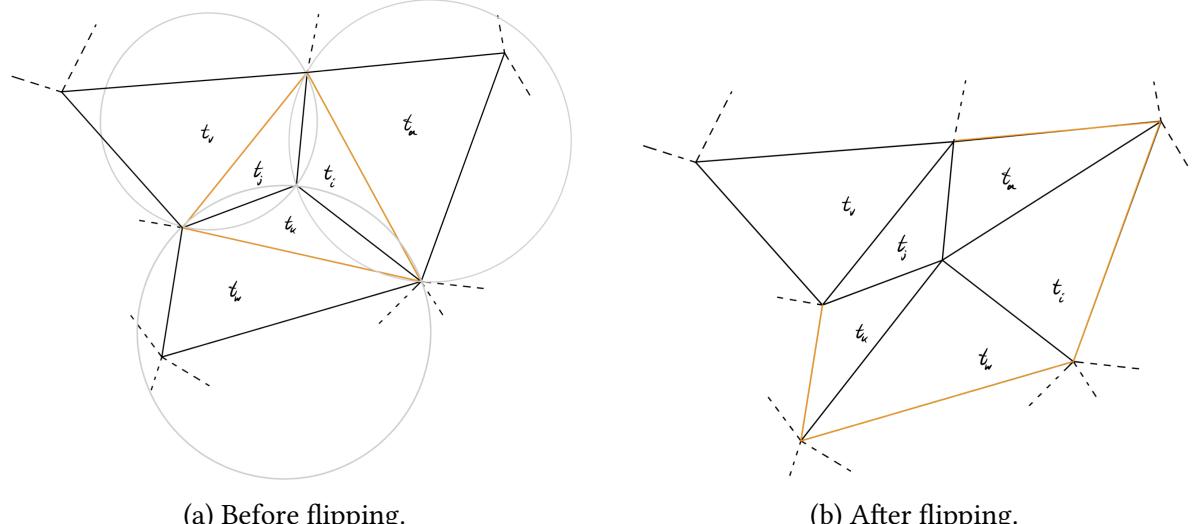


Figure 6: Output from only running the point insertion triangulation algorithm. The additional points added to form the super triangle and triangles containing these points are removed from this uniform distribution of points on a disk.

### 3.1.2. Flipping

Once a point insertion step is complete, appropriate flipping operations are then performed. Figure 7 illustrates this procedure. One can observe that the new edges introduced by the point insertion do not need to be flipped as they their circumcircles will not contain the points opposite the edge by construction [10] and also would interfere with other triangles if flipped as the configurations are not convex. New edges are chosen to be ones which have not been previously flipped surrounding the point insertion and only need to be checked once.



(a) Before flipping.

(b) After flipping.

Figure 7: Illustrating the flipping operation. In figure (a), point  $r$  has just been inserted and the orange edges are have been marked to be checked for flipping. Two of these end edges end up being flipped in (b). The edges inside would not qualify for flipping as any quadrilateral would not form a convex region. In order to perform the edge flipping algorithm we choose to construct the two new triangles which would form after the edge flip and the overwrite the two triangles which should no longer exist.

A useful construction for ease of implementation and readability is to create a temporary *quad* data structure which contains the necessary information for constructing the new triangles. The existing edge can be thought of a being rotated counter clockwise which lets us know where the indexes of the previous triangles are being overwritten to in the array of *tri* structures in later described in Listing 2. Most of the triangles can be constructed internally but the neighbouring triangles also need to have their neighbour information and points are opposite the neigbouring edge updated.

### 3.1.3. Implementation

The implementaion was written in C++ and was not written with a large amount of object oriented programming (OOP) techniques for an gentler transition to a CUDA implementaion as CUDA heavily relies on pointer semantics and does not support some of the more convenient OOP feautres. However as CUDA does support OOP features on the host side so the I chose to write a *Delaunay* class which holds most of the important features of the computation as methods which are exectued in the constructor of the *Delaunay* object.

A quick demo of how to use the object is given below.

```

#include "delaunay.h"
#include "point.h"
#include "ran.h"

int main(int argc, char *argv[]) {

    int n = 100;
    int seed = 69420;

    Point* points = (Point*) malloc(n * sizeof(Point));

    Ran ran(seed);
    for (int i=0; i<n; ++i) {
        points[i].x[0] = ran.doub();
        points[i].x[1] = ran.doub();
    }

    Delaunay delaunay(points, n);

    free(points);
    return 0;
}

```

Listing 1: A quick illustration of how the Delaunay class is called. This code and other relevant source is located in *main/serialIncPtInsertion/src*. Construct an array of points and pass the pointer and the number of pointes generated as arguments to the Delaunay object. A file is created *main/serialIncPtInsertion/data/tri.txt* with the history of and final result of the algorithm.

### 3.1.4. Analysis

The analysis in this section will be brief but I hope succinct as the majority of the work done was involved in the parallelized versions of this algorithm showcased in the following sections.

In Figure 8 below we can observe the time complexity of the serial algorithm. This algorithm can theoretically achieve a complexity of  $O(n \log(n))$  however my naive implementation does not achieve this and we have a  $O(n^2)$  scaling as seen by the straight line in the log plot. Even though this is not the result I have hoped for, this is still a useful piece of code to compare the future GPU implementation with. I believe that a  $O(n \log(n))$  complexity can be achieved by using a directed acyclic graph structure (DAG) for faster memory access in finding in which triangles points are contained in.

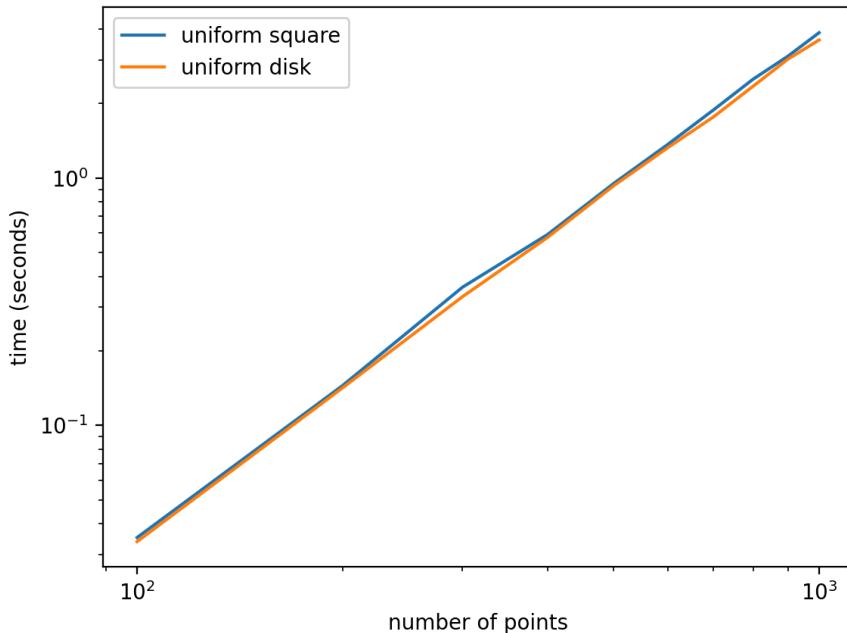


Figure 8: Plot showing the amount of time it took serial code to run with respect to the number of points in the triangulation. This is a loglog plot which shows us the algorithm has a complexity of  $O(n^2)$

### 3.2. Parallel

The parallelization of the DT is conceptually not very different than its serial counterpart. We will be considering only parallelization with a GPU here which lends itself to algorithms which are created with a GPUs architecture in mind. This means that accesing data will be largely done by accessing global arrays which all threads of execution have access to. Methods akin to divide and conquer [11] would be useful if we consider multi CPU or multi GPU systems but that is not in the scope of this project but would be particulary interesing to see a multi GPU systems implementation for this algorithm made publicly available. An overview of the parallelized algorithm is in Algorithm 4 mostly adapted from [12] which is to my understanding as of this moment the fastest GPU delaunay triangulation algorithm.

---

**Algorithm 4:** Parallel point insertion and flipping

---

Data: A point set  $P$   
Out: Delaunay Triangulation  $T$

```
1 Initialize  $T$  with a triangle  $t$  enclosing all points in  $P$ 
2 Initialize locations of  $p \in P$  to all lie in  $t$ 
3 while there are  $p \in P$  to insert
4   for each  $p \in P$  do in parallel
5     | choose  $p_t \in P$  to insert if any
6     for each  $t \in T$  with  $p_t$  to insert do in parallel
7       | split  $t$ 
8     while there are illegal edges
9       for each triangle  $t \in T$  do in parallel
10      | mark whether it should be flipped
11      for each triangle  $t \in T$  in a configuration marked to flip do in parallel
12        | flip  $t$ 
13   update locations of  $p \in P$ 
14 return  $T$ 
```

---

Algorithm 4 takes as input a point set  $P$  for the triangulation to be constructed from and return the DT from the transformed triangulation  $T$ . (line 1) The triangulation is initialized as a triangle enclosing all points in  $P$  by adding 3 new points to the triangulation. These extra three points will later be removed. (line 3) Tells us to keep performing the main work of the algorithm as long as there are points to be inserted into  $T$ . (lines 3-4) We pick out points in parallel which can be inserted into  $T$  by checking in which triangle each point not yet inserted, if any, is closest to the circumcenter of the triangle. This point will be inserted in the (lines\_5-6) in which for every triangle which has a point inside it to be inserted we split the existing triangle  $t$  into 3 new triangles which all contain the inserted point  $p$ . Now in (lines 7-10) at this point, we have a non Delaunay mesh which needs to be transformed and so we perform necessary flipping operations in order for this to be a DT. For each triangle we first check whether we should flip with any 3 any of its neighbours by checking if each edge is illegal. If an edge is found to be illegal the first neighbouring triangle is marked to be flipped with. Following this we check whether any triangles marked for flipping would be conflicting with any other configuration flipping, and if so, it is discarded for this iteration of the while loop. In (lines 11-12) we perform the flipping operation for each triangle which wont have any conflicts. At the end of the outermost while loop in (line 13) we update our knowledge of where points which have not yet been inserted not lie after the changes by the point insertion creating new triangles and flipping changing the triangles themselves.

Algorithm 4 exploits the most parallelizable aspects of the point insertion Algorithm 2, which are the point insertion, for which only one triangle is involved in at a time, and the flipping operation, which can be parallelized but some book keeping needs to be taken care of in order for conflicting flip to not be performed. With a large point set this parallelization allows for a massively parallel algorithm as a large number of point insertions and flips can be performed in parallel. Flipping conflicts can happen when two different configurations of neighbouring triangles want to flip and these two configurations share a triangle, as illustrated in Figure 10.

### 3.2.1. Insertion

The parallel point insertion step is very well suited for parallelization. Parallel point insertion can be performed without any interference with their neighbours. This procedure is performed independently for each triangle with a point to insert. The only complication arises in the updating of neighbouring triangles information about their newly updated neighbours and opposite points. This must be done after all new triangles have been constructed and saved to memory. Only then you can exploit the data structure and traverse the neighbouring triangle to update the correct triangles appropriate edge.

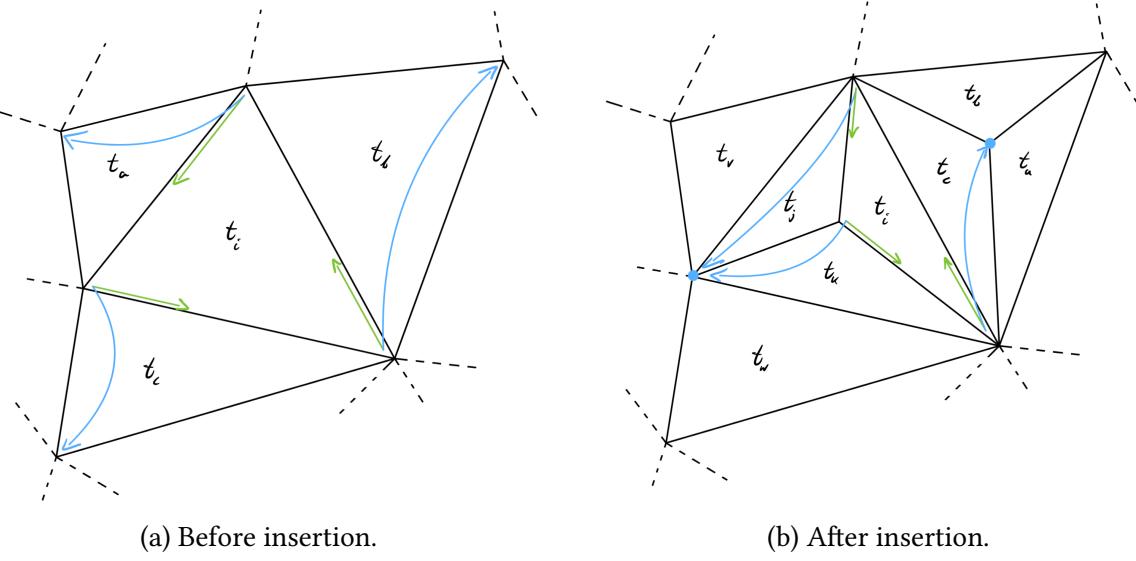


Figure 9: Parallel point insertion

## Implementation

The implementation of the parallel point insertion algorithm relies on two steps, preparation of points to be inserted and the insertion of points. If only the point insertion procedure is performed we also need to update point locations which is normally done after the flipping operations needed.

The preparation step involves a handful of checks or verifications to find out which point should be inserted into each triangle. In this algorithm we wish to find the most suitable point for each triangle to have inserted into it. We do this by finding out which point, which is not yet inserted into the triangulation lies in which triangle. The point closest to the circumcenter of the triangle is chosen to be inserted. Two CUDA kernels are used in this procedure, one to calculate the distances of each point to their corresponding circumcentres and another to find the minimum distance. This procedure relies on computing the distance twice as compute is cheap on GPU as opposed to copying memory of the triangle structures. In between all of this arrays which contain information about uninjected points *ptsUninjected* are used throughout in order to not waste resources in the form of threads which would obtain instructions to do nothing. The *ptsUninjected* array is sorted in order to launch the minimum number of threads needed. A few other kernels are used for book keeping purposes which aren't explained.

Once the preparation step is completed, which makes up the majority of the compute for point insertion procedure (Figure 15) we can now actually insert the points which have been picked out. The logic in the parallel point insertion step is as follows. The logic is mostly consistent as in Figure 5 but needs to be adapted in order for it to be parallelized. For the creation and rewriting involved in making the 3 new triangles stays the same except for the updating of the neighbouring triangles before insertion. If a neighbouring triangle will not perform a point insertion

---

**Algorithm 5:** prepForInsert

---

```
1 updatePtsUninsterted  
2 gpuSort ptsUninserted_d;  
3 setInsertPtsDistance  
4 setInsertPts  
5 prepTriWithInsert  
6 gpuSort triWithInsert_d nTriMax_d;
```

---

---

**Algorithm 6:** insert

---

```
1 insertKernel  
2 updateNbrsAfterInsertKernel  
3 update num tri  
4 update num pts inserted
```

---

### 3.2.2. Flipping

As briefly mentioned earlier, flipping can be performed in a highly parallel manner however some care needs to be taken. The logic within the flipping operation is split up into three main steps. The first one is the reading of triangles to be flipping in the corresponding configuration into a *Quad* data structure which here is mainly created for the purpose of an easier implementation. This *Quad* strut will aid us in constructing flipped configuration. The two new triangles are then written by one kernel and appropriate neighbours are then updated in a separate kernel.

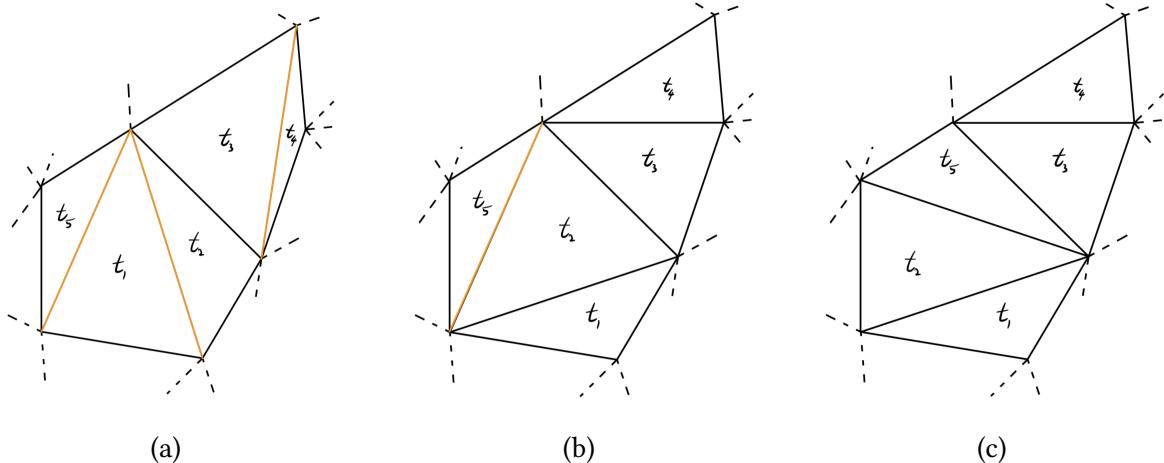


Figure 10: Illustration of parallel flipping while accounting for non-conflicting configurations. Edges colored orange are marked for flipping. For each configuration marked for flipping by each orange edge the triangle with the smallest index will be the one performing the flipping operation, and the configuration with the smallest index (min of both indexes of triangles in the configuration) will have priority to flip first in each round of parallel flipping. In the first figure (a) 3 edges are marked for flipping. Only configurations of triangles  $t_1t_2$  and  $t_3t_4$ , with configuration index 1 and 3 respectively, will flip. Configuration  $t_5, t_1$  with a configuration index of 1 will not flip in the first parallel flipping iteration (b) as it is not the minimum index in its configuration. (c) Showcases the final outcome of the parallel flipping.

### **3.2.3. Implementation Flipping**

### **3.2.4. Analysis**

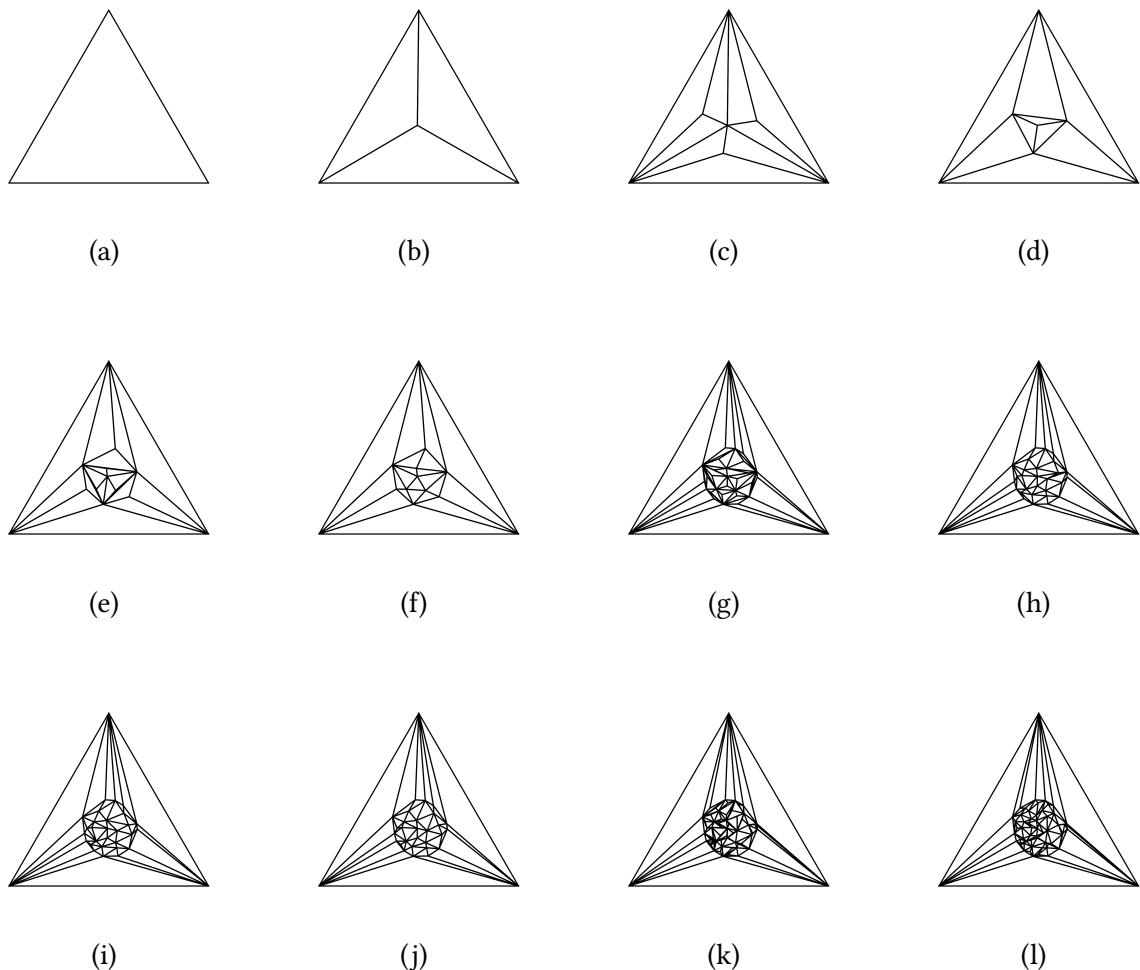
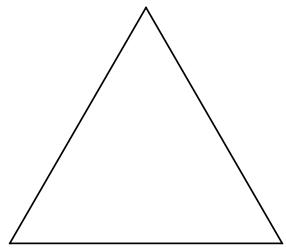
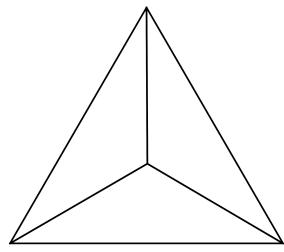


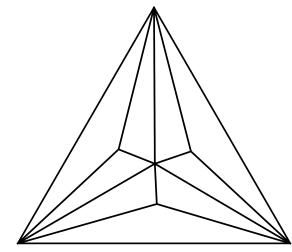
Figure 11: These figures show the history of the DT algorithm. The algorithm begins by initializing a super triangle (a) which is constructed to contain each point desired by the user. Here a uniform point distribution on a unit disk is used. In (b) and (c) a point insertion is performed and in (c) certain edges are marked for parallel flipping for which the result is displayed in (d). The algorithm proceeds in following subfigures with a series of point insertion followed by the required number of parallel flipping operations. In the final result, triangles which contain the initialized supertriangle points are removed and we are left with the desired triangulation as can be seen in Figure 18.



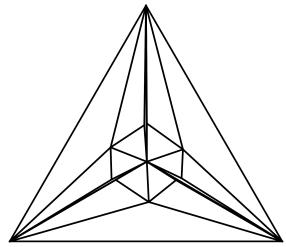
(a)



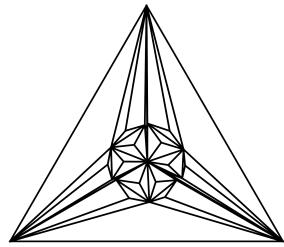
(b)



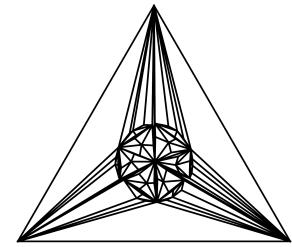
(c)



(d)



(e)



(f)

Figure 12: These figures show the evolution of the only point insertion algorithm. The point insertion proceeds in alphabetical order noting the labels of each subfigure. During the computation points closest to the circumcenter of each triangle are chosen to be inserted and split each existing triangle with a point to insert. These figures use a uniform point distribution on a unit disk.

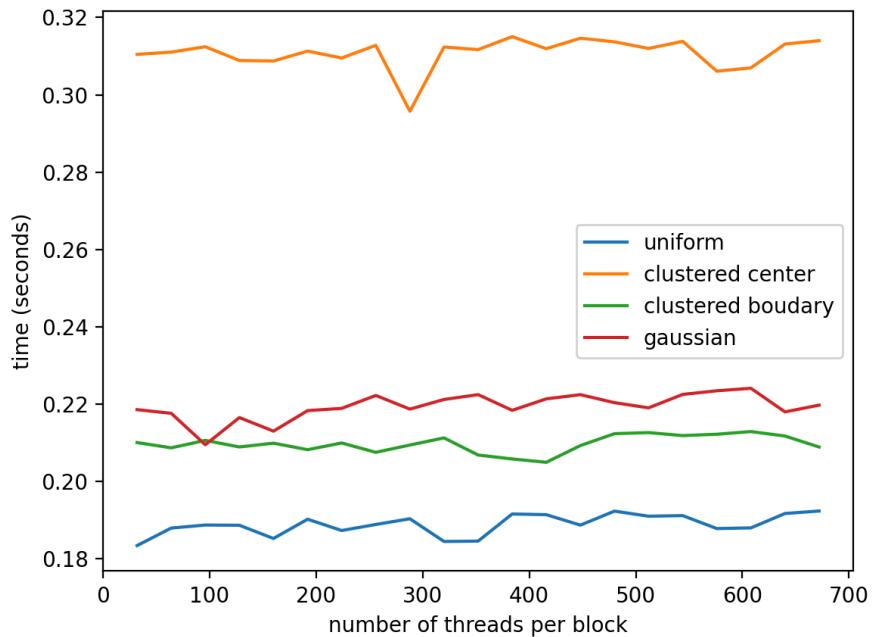


Figure 13: Showing the time it took for the GPU DT code to run with 10000 points while varying the number of threads per block also known as the block size. This is a rather naive way of finding the optimal number of threads per block as a better analysis **TODO** would involve logically similar block of code to have their own block size. Currently the block size doesn't rationally affect the runtime.

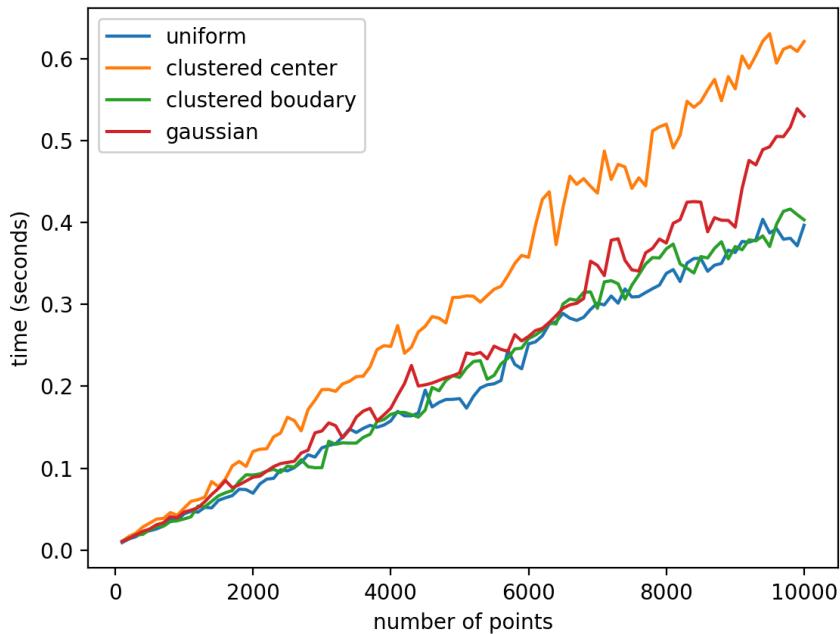


Figure 14: Plot showing the amount of time it took the GPU code to run with respect to the number of points in the triangulation.

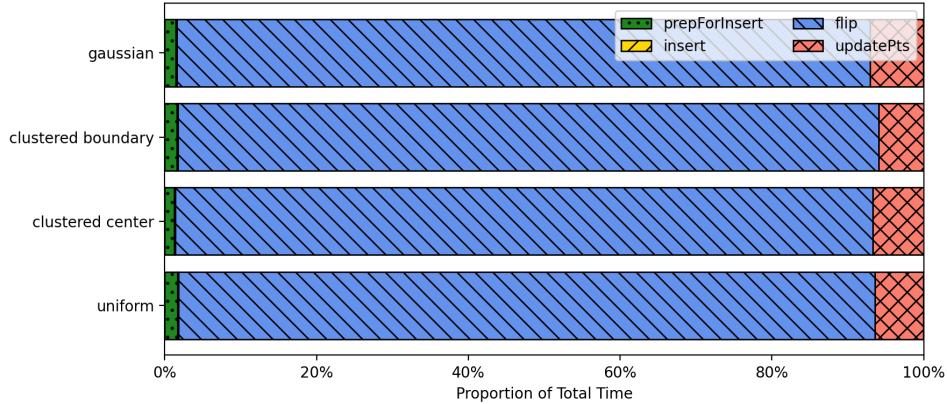


Figure 15: Showing the proportions of time each function took as a percentage of the total runtime.

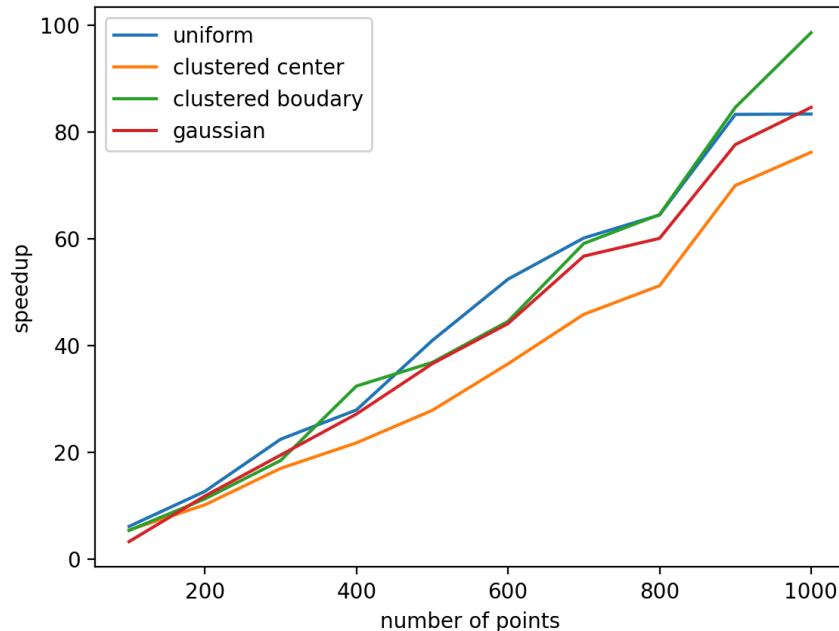


Figure 16: Plot showing speedup of the GPU code with respect to the serial implementation of the incremental point insertion Algorithm 2. The speedup here is comparing the runtime of the serial code with for a given number of points and with the runtime of the GPU code with the same number of points. Both implementations are run with single precision floating point arithmetic.

Speedup here is defined as the ratio  $\frac{\text{timeCPU}}{\text{timeGPU}}$ .

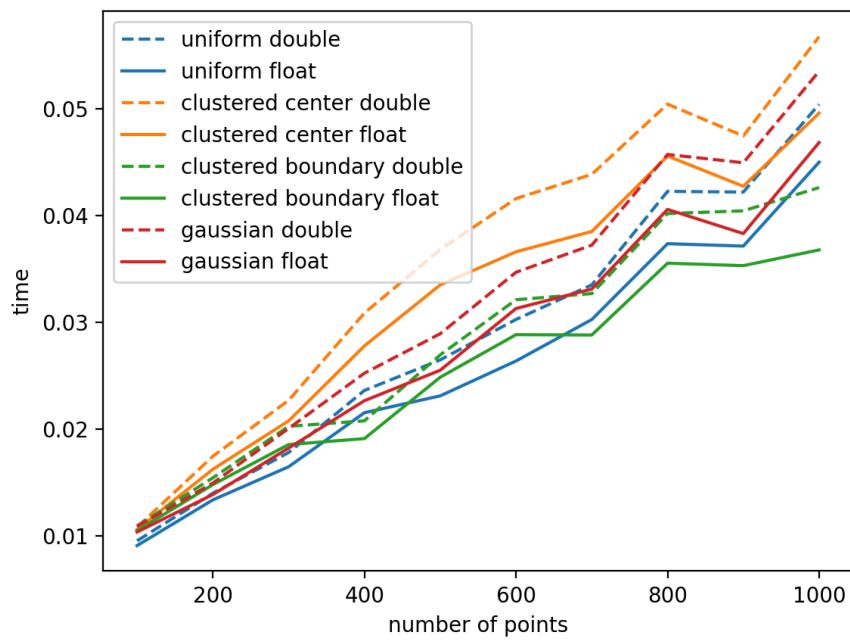


Figure 17: floatVsDouble

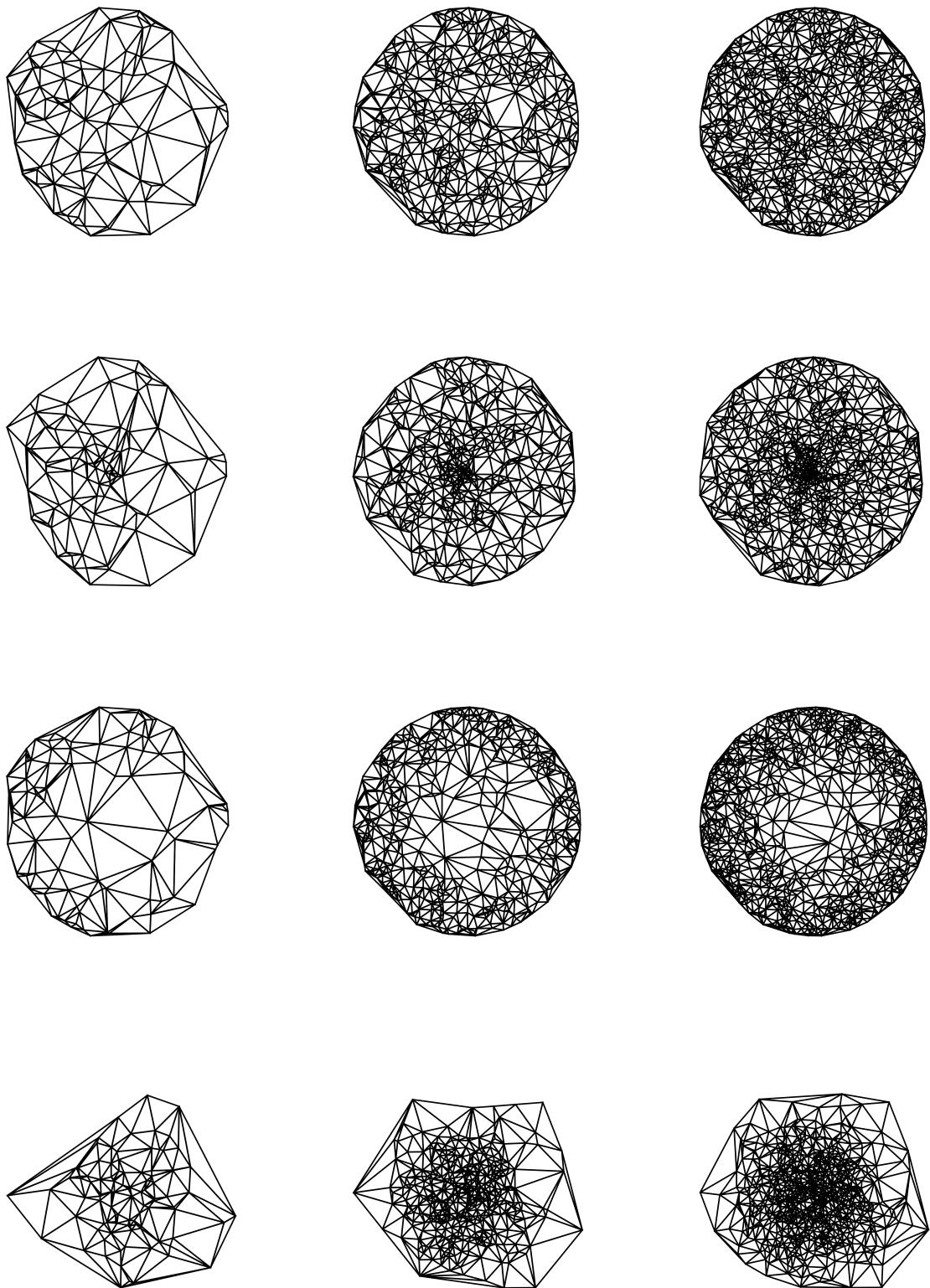


Figure 18: Visualisations of Delaunay triangulations of various point distributions. The grid should be read as follows. Along the horizontal the number of points involved increases gradually and with 100, 500, 1000 points in the the first second and third column respectively. In each row we draw from different point distributions. The rows draw from a uniform unit disk distribution, a distribution on a disk with points clustered in the center, a distribution on a disk with points clustered near the boundary and a gaussian distribution with mean 0 and variance 1, in rows 1, 2, 3 and 4 respectively.

### 3.3. Data Structures

The core data structure that is needed in this algorithm is one to represent a the triangulation itself. There are a handful of different approaches to this problem inculding representing edges by the quaud edge data structure [10] however we choose to represent the triangles in our triangulation by explicit triangle structures [13] which hold neccesary information about their neighbours for the construction of the trianulation and for performing point insertion and flipping operations.

```
struct Tri {
    int p[3]; // points
    int n[3]; // neighbours
    int o[3]; // opposite points
};
```

Listing 2: Triangle data structure. Stores point index info, the indexes of its neighbouring triangles and the points opposite its edges by the index of that point in the corresponding neighbour.

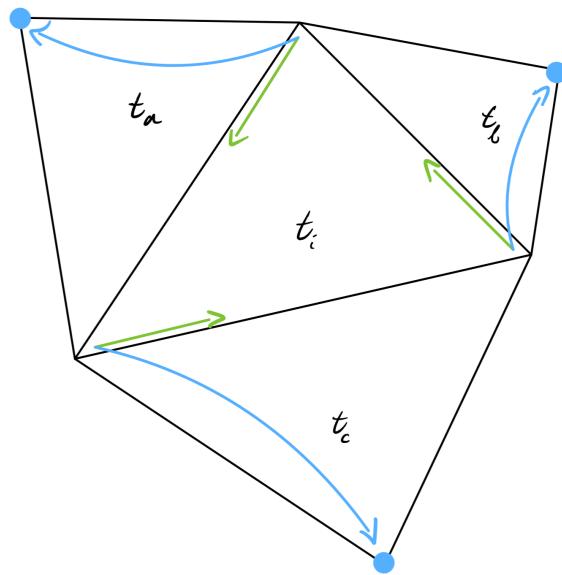


Figure 19: An illustration of the *Tri* data structures main features. We describe the triangle  $t_i$  int the figure. Oriented counter clockwise points are stored as indexes an array containing two dimensional coordinate represeting the point. The neighbours are assigned by using the right hand side of each edge using and index of the point as the start of the edge and following the edge in the CCW direction. The neighbours index will by written in the corresponding entry in the structure.

This data structure was chosen for the ease of implementation and as whenever we want to read a traigle we will be a significant amount of data about it and this locality theoreitcally helps with memory reads, as opposed to storing separate parts of date about the triangle in different structures, ie separating point and neighbour information into two different structs.

The Listing 3 below

```
struct Quad {
    int p[4];
    int n[4];
    int o[4];
};
```

Listing 3: Quad data structure. Stores point index info, the indexes of its neighbouring triangles and the points opposite its edges by the index of that point in the corresponding neighbour.

**3.4. Profiling Analysis**

**3.5. Comment on the Floating point arithmetic and how the maths is funny sometimes**

**3.6. Strong and weak scaling on GPUs?**

## Bibliography

- [1] L. Chen, “Mesh Smoothing Schemes Based on Optimal Delaunay Triangulations,” 2004, pp. 109–120.
- [2] C. L. Lawson, “Transforming triangulations,” *Discrete Math.*, vol. 3, no. 4, pp. 365–372, Jan. 1972, doi: 10.1016/0012-365X(72)90093-3.
- [3] S. L. Devadoss and J. O'Rourke, *Discrete and Computational Geometry, 1st Edition*. Princeton University Press, 2011.
- [4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry: algorithms and applications*. Berlin, Heidelberg: Springer-Verlag, 1997.
- [5] C. Lawson, “Software for C1 Surface Interpolation.” [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B978012587260750011X>
- [6] M. J. Flynn, “Some computer organizations and their effectiveness,” *IEEE Trans. Comput.*, vol. 21, no. 9, pp. 948–960, Sep. 1972, doi: 10.1109/TC.1972.5009071.
- [7] “CUDA C++ Programming Guide.” [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide>
- [8] “Delaunay Triangulations.” [Online]. Available: <https://ti.inf.ethz.ch/ew/courses/Geo23/lecture/gca23-6.pdf>
- [9] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. USA: Cambridge University Press, 2007.
- [10] L. Guibas and J. Stolfi, “Primitives for the manipulation of general subdivisions and the computation of Voronoi,” *ACM Trans. Graph.*, vol. 4, no. 2, pp. 74–123, Apr. 1985, doi: 10.1145/282918.282923.
- [11] P. Cignoni, C. Montani, and R. Scopigno, “DeWall: A fast divide and conquer Delaunay triangulation algorithm in E d,” *Computer-Aided Design*, vol. 30, pp. 333–341, 1998, doi: 10.1016/S0010-4485(97)00082-1.
- [12] T.-T. Cao, A. Nanjappa, M. Gao, and T.-S. Tan, “A GPU accelerated algorithm for 3D Delaunay triangulation,” in *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, in I3D '14. San Francisco, California: Association for Computing Machinery, 2014, pp. 47–54. doi: 10.1145/2556700.2556710.
- [13] A. Nanjappa, “Delaunay triangulation in R3 on the GPU,” 2012.