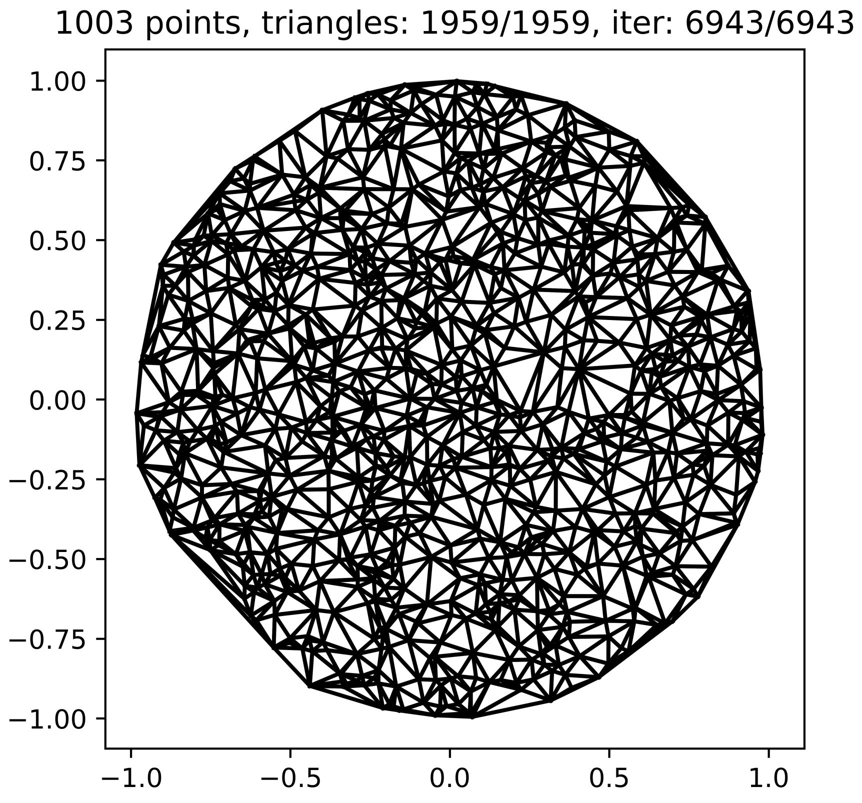
Delaunay Triangulations on the GPU

Patryk Drozd
Trinity College Dublin
drozdp@tcd.ie



Contents

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 a commonly uses for simulations such as compulational fluid dynamics or other physical properties, and even have use in video games for rendiring and visualising complex geometries. To paint a picure 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 instersecting. A particulary 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 triangulatinos in particular are a good candiate 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 alot of them use the 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 optaining the optimal Delaunay triangulation. This flipping operation involves a configuration of two triangles sharing an edge, forming a quadrilateral with its boundary. The shared egde 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 agorithm motivated by ([2]) is hinted to be us this flipping operation to iterate through different triangulations and eventually arrive at the Delaunay trianglution which we desire.

With the flippig operation being at the core of the algorithm, we can notice that is 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 parallize 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 SIMT 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 algorthm designed with 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 indroduce the Delaunay Traingulation we first must define what we mean mean by a triangultion. In order to create a triangualtion 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].

Alterantively we could say an edge doesnt contain its enpoints 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 imortant point in Definition 1.2 is that it is a *maximal* set of noncrossing egdes which for us means that we will not have any other shapes than triangles int this structure.

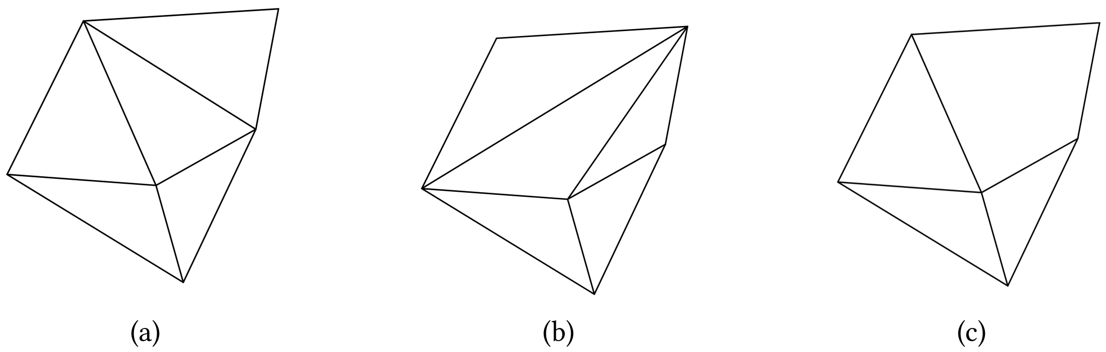


Figure 2: Examples of two traingulations (a) (b) on the same set of points. In (c) an illustration of a non maximal set of edges.

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 allways be a set which will covering a set of points, in our case the points in our triangulation. This will be usefull when we will be storing triangles as we will allways 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 alse that no 4 points like on a circle. Motivation for this defintion 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 allways assume that the point set P from which we obtain our triangulation will be in *general position*. This is neccesary for the definitions and theorems we will define.

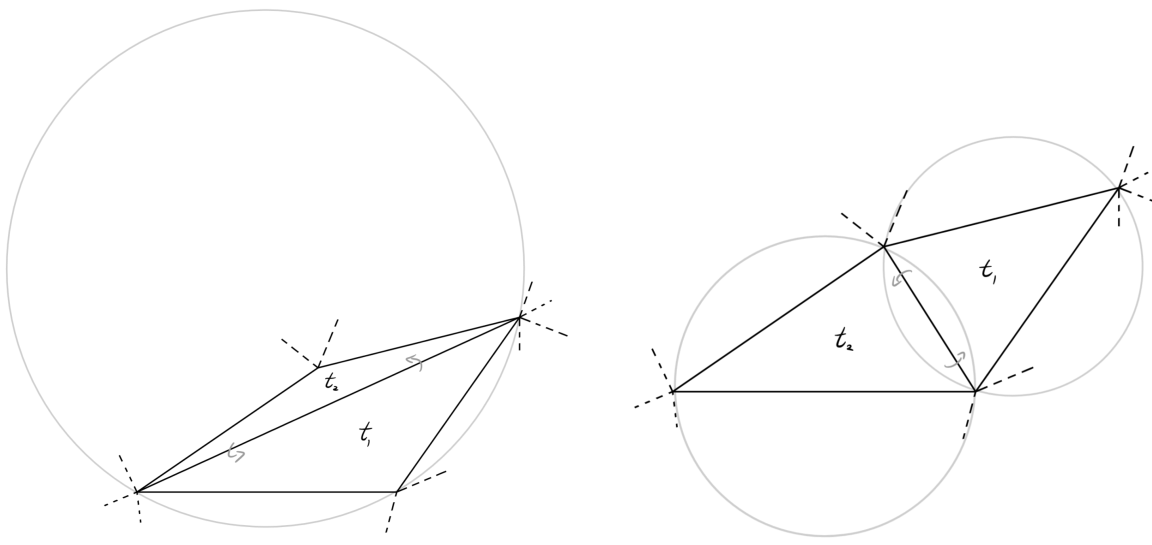
In order to define a Delaunay triangulation we would like to establish the motivation for the definition with another, preliminary definition. A Delaunay triangulation is a type of triangulation which in a sense maximizes smallest angles in a triangulation T. This idea is formalized by defining an *angle sequence* $(\alpha_1, \alpha_2, ..., \alpha_{3n})$ 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 eachother. 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 lexographically 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” triangluation 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]



(a) (b)

Figure 3: Demonstration of the flipping operation. In (a) A configuration that needs to be flipped illustrated by the circumcircle of t_1 containg 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 circumcirles 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 circumcicle 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 intruduced 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 triangluation by stating that we can travel across all possible triangultions 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 such for example in video games through visualizing the two or three dimensional environments the "gamer" would be interacting with or rendering vidoes in movies. 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 mainly lead by Nvidia and AMD, with intel now recently entering the GPU market with their Arc series. Today, the GPU has gained a more general purporse status with the rise of General Purpose GPU (GPGPU) programming as more and more people have notices that GPUs are very useful as a general hardware accelerator.

The triadional CPU (based on the Von Neumann architecture) which is built to perform *serial* tasks with helful features such as branch predicion, for dealing with if statements and vairable lenght instructions like for loops with variable lengh, 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 conjuction to the CPU. The CPU sends compute intructions for the GPU to perform and data is commonly passes between the CPU and GPU.

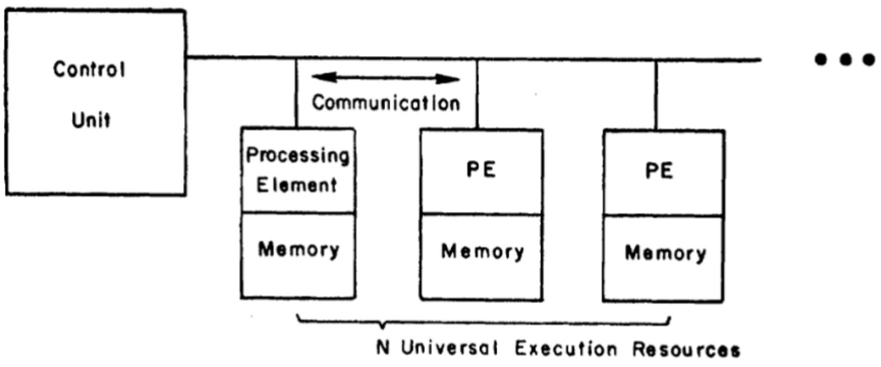


Figure 4: The *Single Instruction Multiple Threads (SIMT)* classification, originally known as an *Array Processor* as illustrated by Michael J. Flynn [6]. The control unit communicates instructions to the N processing element with each processing element having its own memory.

What makes the GPU increadibly usefull in certain usecases (like the one of this report) is its architecture which is build 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 proccessing 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 proccessing of data.

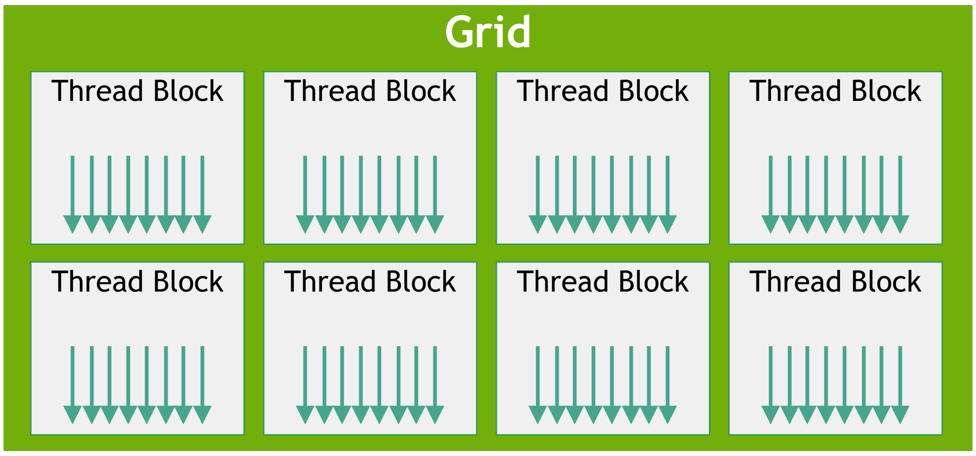


Figure 5: An illustratino of the structure of the GPU programming model. As the lowest compute instrcution we have a thread block consisting of a number threads ≤ 1024. The thread blocks are contained in a grid. [7]

Nvida's GPUs take the SIMT model and further develop it. There are three core abstractions which allow Nvidia's GPU model to be succesfull; a hierarchy of thread groups, shared memories and synchronization [7]. The threads, which represent the a theoretical proccesses which encode programmed instrcutions, 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 organising 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 focus on the parallel algorithm. Commonly algorithms are first developed with a serialized verion and only later optimized into parallelized versions if possible. This is how I will be presenting my chosen Delanay Triangulation algorithms in order to portray a chronolgical development of ideas used in all algorithms.

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 trianulation 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 traingultion is legal and if not flipping it. Most DT algorithms take this core concept and build a more optimized verions 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 incrimental 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 implementaion. The construction this algorithm is a bit mathematically involved however the motivation behind the construction of the algorithm is to perform point instertions, 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 runtume complexity.

Algorithm 2: Randomized incremental point insertion

```
Data: point set $P$
    Initialize $T$ with a triangle enclosing all points in $P$
    Compute a random permutation of $P$
    for $p \in P$
    Insert $p$ into the triangle $t$ containing it
    for each edge $e \in t$
        LegalizeEdge($e$, $t$)
        return $T$
```

A signficant part of this algorithmis 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.

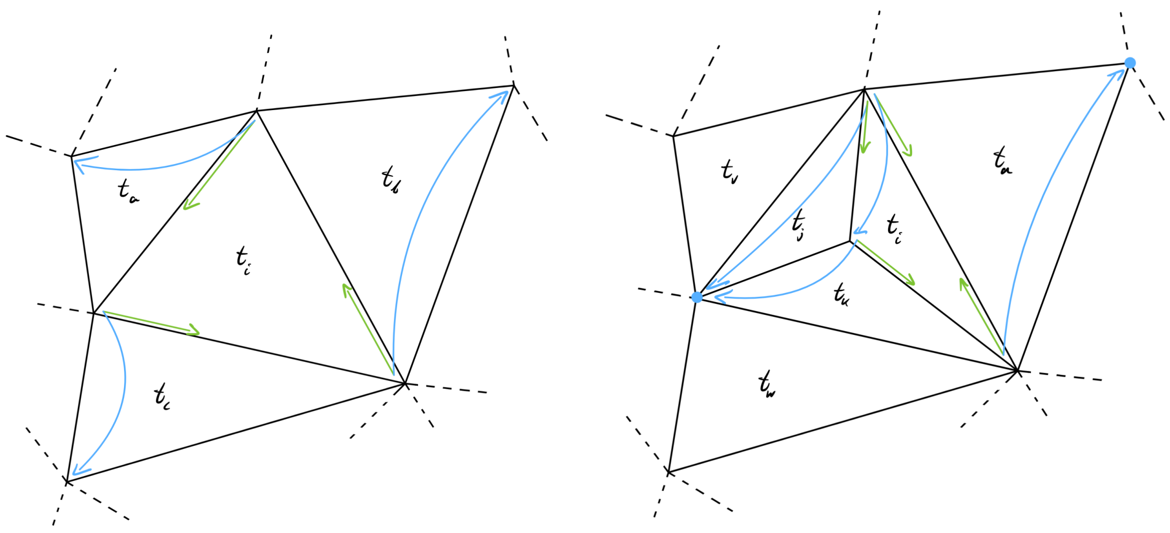
Algorithm 3: LegalizeEdge

```
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($e_2$, $t_b$)
```

In the following sections we will discuss the point insertion and flipping steps in more detail.

3.1.1. Point insertion

In the point instertion procedure there a point not yet insterted in the point set making up the final triagulation is chosen and then the triangle in which it lies in is found and selected for insertion. Figure 6 illustrates this process.



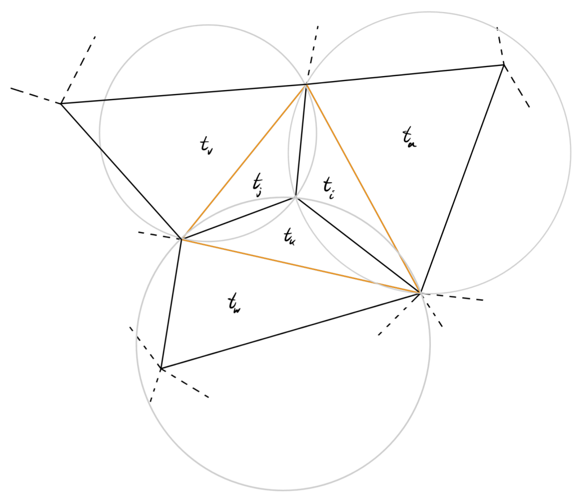
(a) Before insertion.

(b) After insertion.

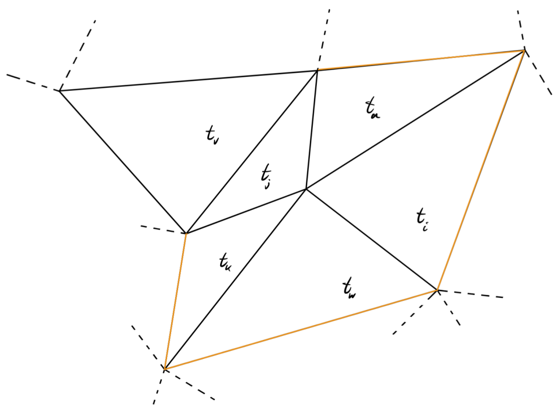
Figure 6: An illustration of the point insertion in step 4 of Algorithm 2. In figure (a) the center most triangle t_i will be then triangle in which a point will be chosen for insertion. Triangle t_i knows its neighbours across each edge represented by the green arrows and knows the points opposite each 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.

3.1.2. Flipping

Once the point insertion step is complete, appropirate 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 [10].



(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.

3.1.3. Implementation

The implementaion was written in a C++ style, without excessive use of object oriented programmingn (OOP) techniques for an gentler trinsition to a CUDA implementaion as CUDA heavily relies on pointer semantics and doesnt not support some of the more convenient OOP. 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.

Output

3.1.4. Analysis

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

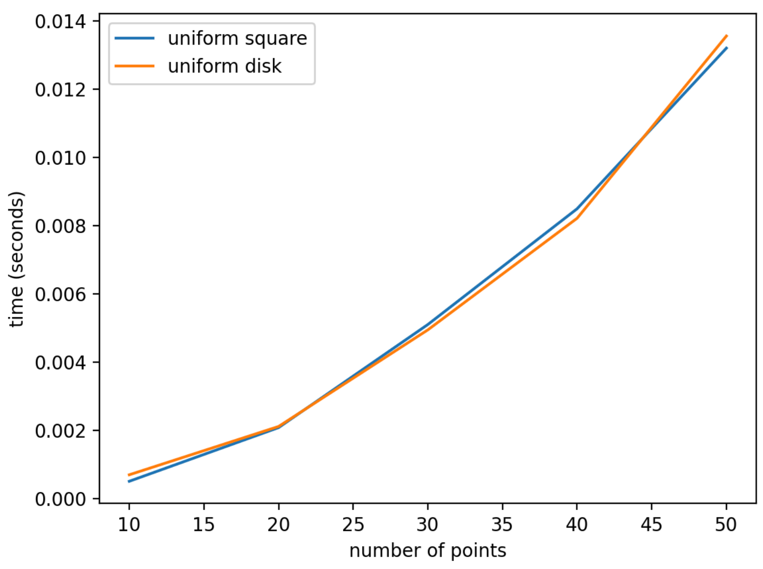


Figure 8: Plot showing the amount of time it took serial code to run with respect to the number of points in the triangulation.

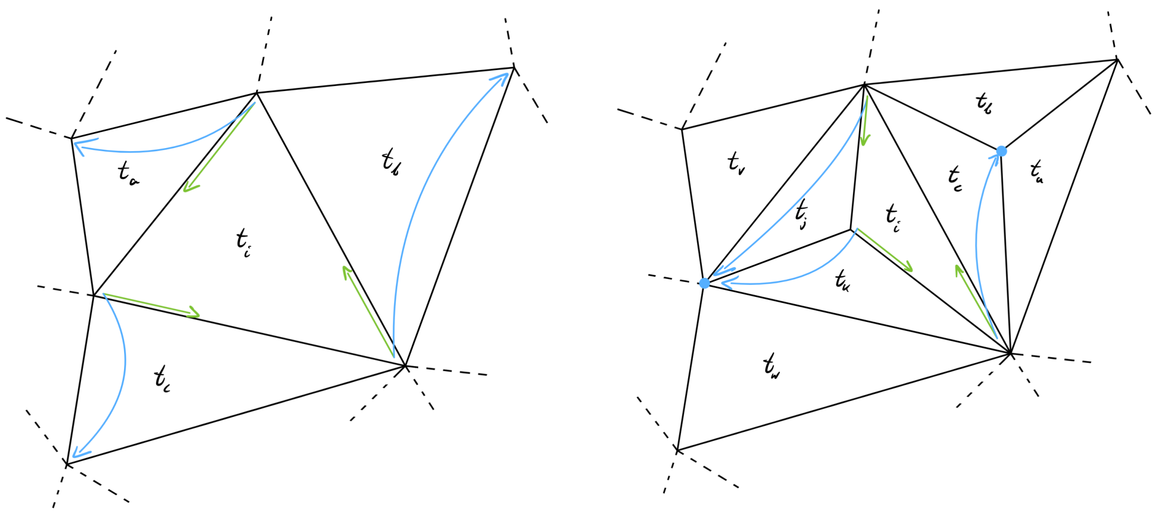
3.2. Parallel

Algorithm 4: Parallel point insertion and flipping [11]

```
Data: A point set $P$
Initialize $T$ with a triangle $t$ enclosing all points in $P$
while there are $p \in P$ to insert
    for each $p \in P$ do in parallel
        choose $p_t \in P$ to insert
    for each $t \in T$ with $p_t$ to insert do in parallel
        split $t$
    while there are configurations to flip
        for each base triangle $t \in T$ in a configuration marked to flip
            flip $t$
    update locations of $p \in P$
return $T$
```

3.2.1. Insertion

The parellel point insertion step is very well suited for parallelization. Without the exect methods of implementation, parallel point insertion can be performed without any interfernce with their neighbours. This procedure is performed independantly 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 a update the correct triangles appropirate edge.



(a) Before insertion. (b) After insertion.

Figure 9:

3.2.2. Flipping

3.2.3. Implementation

This parallelized version of the Incremental Point Instertion alogorithm Algorithm 2, as showcased in Algorithm 4, is written in CUDA.

Layout

The majoriy of the code is wrapped in a *Delaunay* class for the ease of readability and maintainability. The constructor once again performs the computaion however this time on any available GPU. This OOP approach was chosen because of the

Insertion

The parallel point insertion proceduce is implemented as two distinct operations. The *PrepForInsert* method performs key steps to prepare the triangulation for the parallel insertion of new points. This method

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 updateNbrsAfterIsertKernel
3 update num tri
4 update num pts inserted
```

and following it the *insert* method

Flipping

3.2.4. Analysis

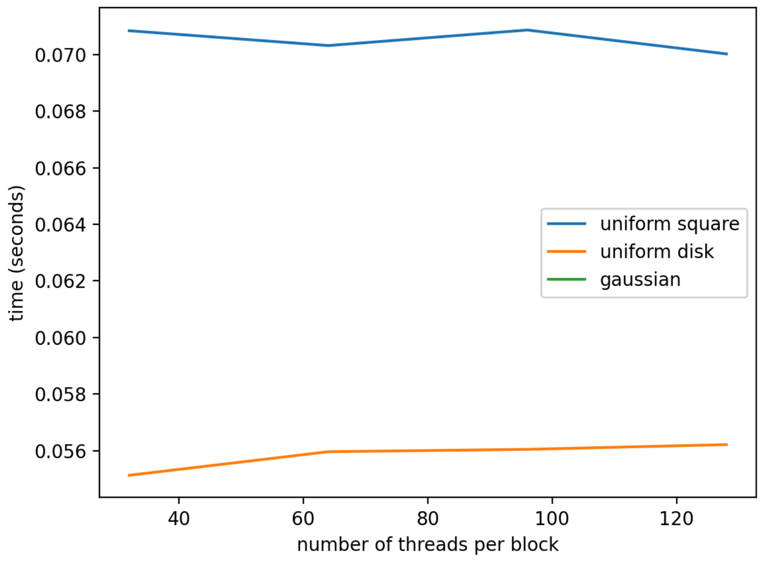


Figure 10: Analysis on finding the best block size to be use by all kerels.

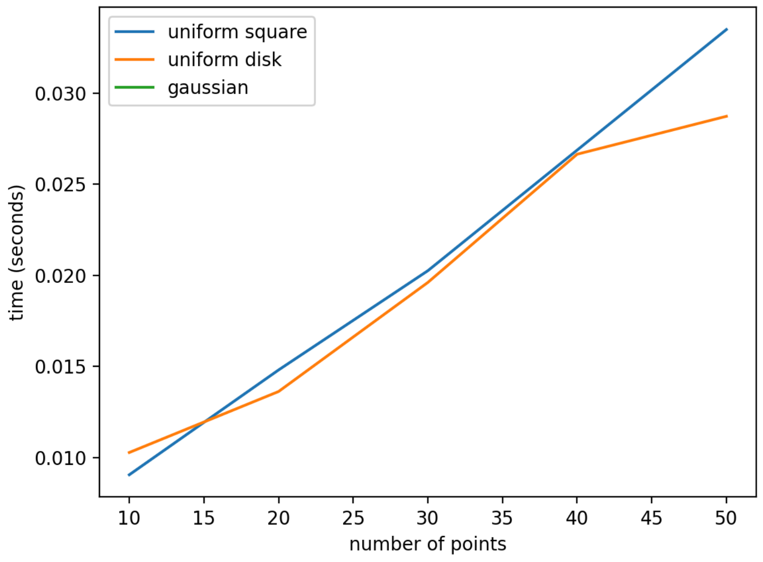


Figure 11: Plot showing the amount of time it took the GPU code to run with respect to the number of points in the triangulation.

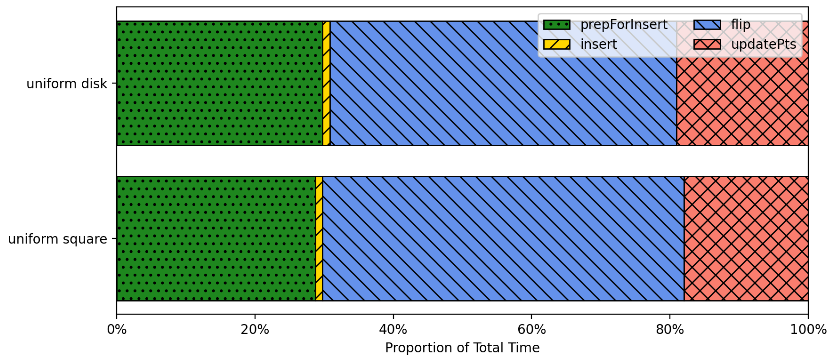


Figure 12: Showing the proportions of time each function took as a percentage of the total runtime.

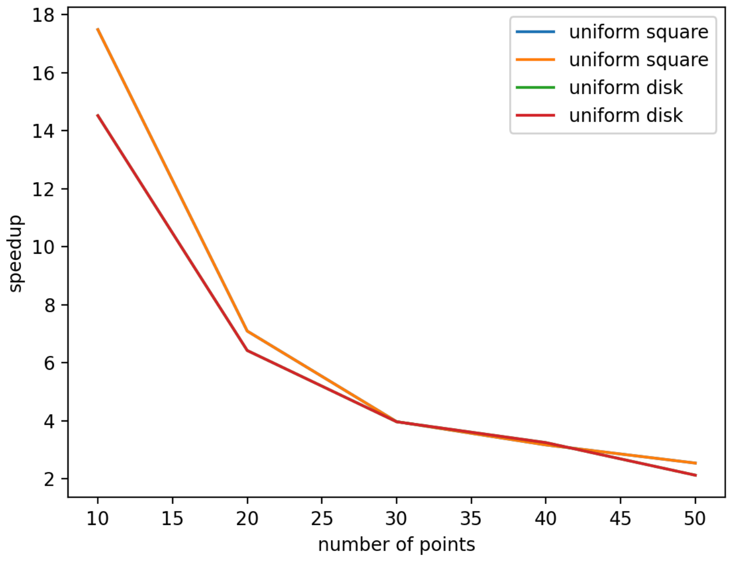
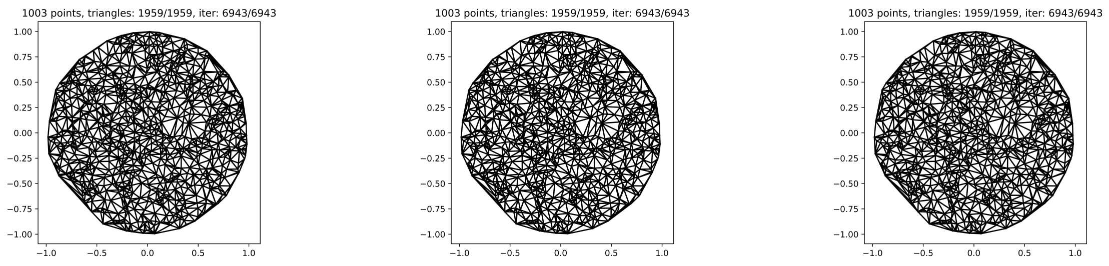


Figure 13: Plot showing speedup of the GPU code with respect to the serial implemenation of the incremental point insertion Algorithm 2. The speedup here is comparing the runtime of the serial code with for a given number of points and with the runtime of the GPU code with the same number of points. Both implementaions are run with single precision floating point arithmetic. Speedup here is defined as the ratio $\frac{\text{timeGPU}}{\text{timeCPU}}$.



1003 points, triangles: 1959/1959, iter: 6943/6943
1003 points, triangles: 1959/1959, iter: 6943/6943
1003 points, triangles: 1959/1959, iter: 6943/6943

(a) Uniform distribution of unit square.

(b) Uniform distribution of unit Disk.

(c) Gaussian distribution with mean 0 and variance 1.

Figure 14: Visualisations of Delaunay triangluations of various point distributions.

3.3. Data Structures

3.3.1. Triangles

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 qaud edge data structure [10] however we choose to represent the triangles in our triangulation by explicit triangle structures [12] 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]; // neighours
  int o[3]; // opposite points
};
```

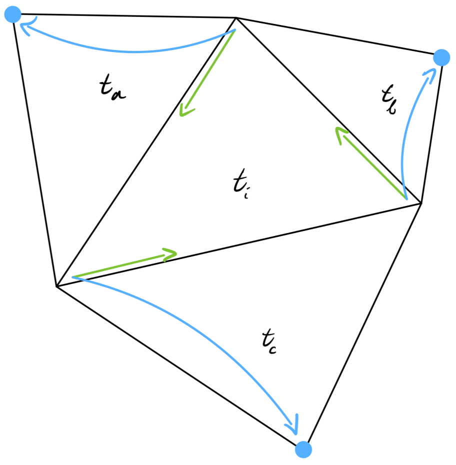


Figure 15: An illustration of the *Tri* data structures main features. We describe the triangle t_i int the figure. Oriented counter clockwise points are stored as indexes an array containing two dimensional coordinate represeting the point. The neighbours are assigned by using the right hand side of 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.

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] 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.

[12] A. Nanjappa, “Delaunay triangulation in R3 on the GPU,” 2012.