UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Implementing High-Performance Delaunay Triangulation In Java

Erik Thune Lund
Department of Informatics, University of Oslo

Masteroppgave høsten 2014

Preface

I would like to thank my supervisor, Arne Maus, for his extreme patience in helping me complete this project.

Contents

1 Introduction **4**

2 Overview of This Report **6**

3 Background **9**

4 Algorithms **11**
- 4.1 Delaunay Triangle Properties 11
  - 4.1.1 The Circumcircle Property 12
  - 4.1.2 The Closest Neighbours Property 13
- 4.2 Testing a Triangle for Validity 14
- 4.3 Standard Search Triangulation 15
- 4.4 The Convex Hull 15
- 4.5 Closest Neighbours of a Point 16
- 4.6 Radial Triangulation 16

5 Data Structures **19**
- 5.1 The Set of Points 19
  - 5.1.1 Point As a Class 19
  - 5.1.2 Interleaved Array of Components 20
  - 5.1.3 Seperate Component Arrays 20
- 5.2 Sorting and Partitioning the Set of Points 21
  - 5.2.1 Sorting 21
- 5.3 Partitioning 22
  - 5.3.1 Segment Partitioning 22
  - 5.3.2 Volume Partitioning 24
  - 5.3.3 Hierarchical Segment Partitioning 25
  - 5.3.4 Grid Partitioning 25
- 5.4 The Convex Hull 27
  - 5.4.1 Querying the Hull for Points 28
  - 5.4.2 Traversal of the Hull 28

6 Implementation **29**
6.1 Data Structures 29
6.1.1 The Set of Points as Component Arrays 29
6.1.2 The Grid Partitioner and Sorting 30
6.1.3 The Convex Hull 33
6.2 Algorithms 33
6.2.1 Nearest Neighbours by Standard Search 33
6.2.2 Nearest Neighbours by Distance Sorting 35
6.2.3 The Convex Hull by Quickhull 35
6.2.4 Triangulation by Standard Search Radial Triangulation 37
6.2.5 Triangulation by Nearest Circumcenter Radial Triangulation 39
6.2.6 Triangulation by Existing Edges Radial Triangulation 39
6.3 Thread Labour Division 39

7 Results **40**
7.1 Data Structures 40
7.1.1 Data Types, *Integer* vs *int* 40
7.1.2 Point As a Class 41
7.1.3 Interleaved Array of Components 41
7.1.4 Seperate Component Arrays 42
7.1.5 Caching Behaviour in PointSet Alternatives 42
7.2 Algorithms 42
7.2.1 The Convex Hull by QuickHull 42
7.2.2 Closest Neighbours 43
7.2.3 Triangulation 44
7.2.4 Algorithm Performance 45
7.3 Program Features 46

8 Conclusion **48**
8.1 Goals 48
8.2 Implementing in Java 49
8.3 Further Work 49

Chapter 1

Introduction

> "Det skal lages en effektiv (tidsmessig rask og med kapasitet til mange datapunkter) i Java av Delaunay-triangulering av n punkter i planet, og det skal lages en meget enkel grafisk modul for å fremvise en slik triangulering på skjerm. I denne modellen skal ulike triangulerings-algoritmer kunne testes ut slik at først bestemmes et større antall (over 50%) av kantene i trianguleringen ved de k nærmeste naboene til hvert punkt (se Maus, Moen Drange: «All closest neighbours are proper Delaunay edges generalized, and its application to parallel algorithm» Nik 2010, Bodø). Ut fra dette fullføres så Delaunay-trianguleringen med minst to andre algoritmer og deres godhet skal vurderes ut fra tidsforbruk. Den grunnleggende algoritmen basert på naboer skal implementeres både som en sekvensiell og en parallell algoritme, og om mulig også de andre algoritmene som undersøkes både få en parallell og sekvensiell utforming. Til parallellisering kan PRP-systemet nyttes (som automatisk parallelliserer en rekursiv løsning). Modellen skal utformes slik at det for senere studenter blir relativt enkelt å bygge ut med algoritmer for trekking av høydekoter, volumberegninger o.l ."

Over the years, it has become increasingly important to model physical systems with greater accuracy, especially terrain maps and in simulations. This can require a very large number of spatial data points which are often triangulated to create a smooth mesh for further processing. One of the primary, if not the primary method of triangulation is Delaunay triangulation. Algorithms have been developed previously that can create the Delaunay triangulation in $O(n \log \log n)$ time, but as the amount of data increases even these become too slow. To remedy this, parallel algorithms have been developed that can exploit the high degree of parallelism and concurrency in modern computer architectures.

The project is all about the design, documentation and implementation of a program to perform Delaunay triangulation. The program is written in Java and is focused on performance both with regard to run-time and memory usage. A goal for the program is to triangulate at least 10 million points within a reasonable amount of time. The program is designed for the development and testing of new sequential and parallel algorithms and data structures to be used when performing Delaunay triangulation.

The documentation in this report and the source code are meant to be used for understanding the program so that it may be used as an example in future efforts or to be extended with new capabilities. The program can create the Delaunay triangulation of the set of points, P, which contains n points in a plane by using a variety of algorithms and data structures. An important goal of the project is to explore the ability to parallelize the process on a modern computer architecture and this is noted where appropriate.

Some of the conclusions in short are that Delaunay triangulation is a problem that benefits greatly from parallelization with up to 5x speedups on an 8-core(4 physical cores) CPU. The behaviour of the Java JIT-compiler is an important factor to take into account when writing a high-performance Java program. The importance of CPU-cache should not be underestimated in the design of data structures and algorithms on modern architectures and finally that triangulating 10 million points requires well within the range of reasonable time(Between 15-20s on the test machine).

Chapter 2

Overview of This Report

Chapter 1 is an introduction to the project. It briefly describes the goals of the project and some of the content in the report as well as a summary of the conclusions drawn.

Chapter 2 is this section that describes each chapter in brief and what can be expected in each of them.

Chapter 3 outlines some of the history behind Delaunay triangulation. How the triangulation has been done, what it has been used for and how this has changed over the years. Moving up to more modern history it describes some of the approaches and breakthroughs that have been made.

In chapter 4 the theory of Delaunay triangulation, the various properties of the triangulation and the algorithms used in the program are described. The basic properties of the triangulation are described and how they are used to determine the triangulation. Several algorithms describing how to create the triangulation of P are outlined, some of which are implemented in the program. They are described in detail and later evaluated in terms of their efficiency, with regard to run-time, memory requirements and overall complexity.

In chapter 5 I discuss the data structures that the algorithms use. The goals and requirements of each data structure is outlined and alternatives for each discussed. There are many alternatives and several of the most promising or most practical ones are described in their own section. The data structures are too evaluated in terms of efficency, specifically in how they impact run-time performance and memory usage.

In Chapter 6 the structure of the program is then described, outlining the different classes and modules of the program and how they interact. Explanations of design decisions are given as well as information on where it is

possible to extend the program. All of the internals of the program, including the several common steps done, how the algorithms and data structures fit in, the alternative capabilities and several other specific details.

When discussing the results of the implementation, each data structure is given its own section. Here the exact implementations of each data structure is discussed as well as the measures taken to ensure good performance. Testing is also done to describe the performance profile of each structure and to compare alternatives. Of interest is also how well these data structures perform in a multi-threaded environment and which, if any precautions have been undertaken to ensure thread-safety.

Following the data structures, the implementations of the algorithms are described in similar detail. All of the measures and techniques used to ensure that the algorithms both perform well and give a valid triangulation are explained. The algorithms are compared against each for different values of n and evaluated based on their run-time and memory requirements. Since all the algorithms can run as either sequential or parallel algorithms, the precautions in the implementation taken to ensure this is possible are also described.

After explaining how the program works and all its internals, the results are analyzed and presented in chapter 7. This includes how the different data structures and algorithms performed as well as possible reasons for why. There are sections on the various configurable parameters in the program, mostly those pertaining to to run-time and how they impact it. Of particular interest here is how well the program scales to multiple cores with different parameters, algorithms and data structures.

Problems during implementation, both performance trade-offs and bugs are listed and described in as much detail as they are known. Possible fixes for some of the bugs are described as well as reasons for why they were not implemented.

In the final chapter, chapter 8, some conclusions are drawn about how well the goals have been met. What kind of performance profile that has been achieved, whether the program is suitable to used for further learning, if it can be extended to accommodate new algorithms and data structures easily and what can be expected from it in general.

Some comments are also made on the experience of programming high performance Java and what measures had to be taken to avoid some of the performance problems that can occur.

Finally there are some thoughts on how the program could be improved. Which algorithms or data structures that might be a benefit but were not

tested during the course of project and which bottlenecks should be solved to give the greatest performance increases.

Chapter 3

Background

Delaunay triangulation is a method for finding the Delaunay triangles of a set of points, in this case in a plane. Delaunay triangles are defined as the triangles of a set P of n points where three points from P form a triangle and the circumscribed circle of each triangle contains no points from P internally. Every point in P lies on the circumference of such a circumcircle. These triangles create a connected graph of all the points, leading to the smallest angle in the set being maximized, resulting in the mesh of triangles having few skinny triangles. Since the mesh has few skinny triangles, this can be a benefit for processing in applications by humans and machines alike.

Delaunay triangulation was first written about by Boris Delaunay in 1934[1] when he devised the method as a form of triangulation. It is most often used to model a surface defined by P, which is a set of N-dimensional points. While the triangulation can be generalized to N dimensions, the most common ones to be used are 2- and 3-dimensional triangulations.

The triangulation itself has, in the past, been an $O(n^3)$ procedure which may involve amongst other things: determining the convex hull of the points, performing many costly trigonometric or square-root calculations to determine their attributes, and so on. A number of algorithms have been developed of varying efficiency to deal with the problem(from $O(n^2)$ to $O(n\ log(n))$) which use methods to sort the points, search only the closest neighbours and other techniques. This report will look at ways to efficiently parallelize the process and make use of the Closest-Neighbours property as a way of efficiently dealing with triangulation.

Delaunay triangulation, particularly the 2-dimensional kind is often used to create a surface out of a given set of points, most often as a means to create a mesh, such as for modelling a geographic surface or the surface of an object. The triangulation can also be used for more abstract problems such as finding the Euclidian minimum spanning tree.

Delaunay triangulations can also be used as the starting point for several other processes. Examples of these include point density estimates, search heuristics and the base for mesh refinement.

Chapter 4

Algorithms

The algorithms used to perform the triangulation are equal in importance to the data structure they operate on, if either one has bad performance as n increases, the total performance will suffer greatly. This chapter will present a number of different algorithms and properties and how they can be used to perform Delaunay triangulation.

The differences in the algorithms are important because while all of them will work, they are scalable to different degrees with regard to several factors including, but not limited to: run-time, memory consumption and ability to be parallelized.

A number of different algorithms have been developed to perform Delaunay triangulation. For example: the standard search algorithms that exhaust the search space, incremental algorithms that add poionts and iteratively "flip" invalid triangles, recursive algorithms that grow a triangulation outward from a starting point, radial algorithms that fan around each point and create a "flower" of edges, sweeping algorithms that sweep along an axis and create triangles through converging edges, divide and conquer algorithms that may use any of these techniques but apply them on mergeable subsets. The runtimes of these algorithms vary from $O(n^3)$ to $O(n \, log \, log \, n)$ and some can be parallelized more easily than others.

A number of these algorithms have been implemented using a some of the provable properties in Delaunay triangulation. The algorithms have been implemented to perform both sequentially and in parallel where possible.

4.1 Delaunay Triangle Properties

There are a number of consistent properties that make determining the Delaunay triangulation or finding individual Delaunay triangle edges easier.

Chiefly among these is the circumcircle property and amongst the others it has been found that the Delaunay triangulation is a superset of both the convex hull, the nearest neighbour graph and the Gabriel graph[2]. All of these are relatively easy to find (the two first are $O(n\ log\ n)$ operations) and can be used to jumpstart the triangulation through finding a number of "easy" edges.

4.1.1 The Circumcircle Property

Chief among the ways to determine the Delaunay triangulation is to use the circumcircle property. It reads that a cirle circumscribing any triangle of points from P which does not contain any points from P in its interior is a Delaunay triangle. Using this property, every edge in a triangulation can be verified as a correct Delaunay triangle. See figure 4.1 for some illustrative examples of simple triangulation.

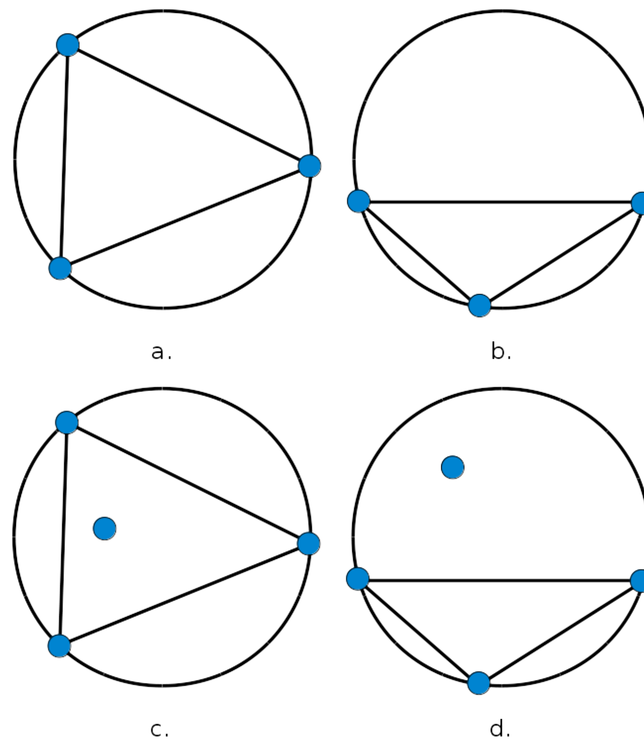

Figure 4.1: *a.* and *b.* show two valid Delaunay triangles and their circumcircles. *c.* and *d.* show two invalid Delaunay triangles because their circumcircles contain another input point. It makes no difference if that point is inside the Delaunay triangle, just the circumcircle.

Edge cases to this property include the possibility of having $n_{co} > 3$ cocircular points. In those cases, an arbitration must be made to divide the n_{co}-gon into an appropriate number of triangles. Figure 4.2 shows an example how to arbitrate the triangles in a case such as this.

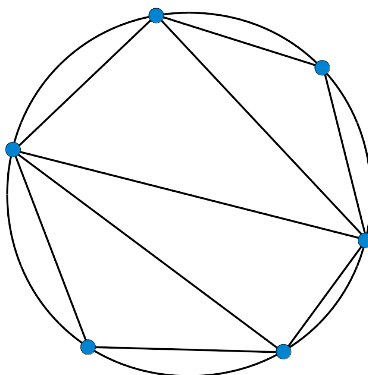

Figure 4.2: The n_{co} cocircular points are sorted by distance from $0, 0)$. Then starting with the first three points p_1, p_2 and p_3, these three points are connected with edges. Then the index for each point is incremented giving the points p_2, p_3, and p_4. These points are connected again and the index increment is repeated. This repeats until the third point of the three is $p_{n_{co}}$, after those final edges have been added, the cocircular points have valid Delaunay triangles.

4.1.2 The Closest Neighbours Property

As described earlier, the Delaunay triangulation is a superset of the Gabriel graph[2]. The closest neighbours property is an extension of this that says any two points from P whose antipodal circle contains no points from P has a Delaunay edge running between them. Figure 4.3 shows this test in practice. This property has been explored earlier[4] and applied to Delaunay triangualtion algorithms.

The Convex Hull Property

The convex hull is usually the first step in determining the Delaunay triangulation as it can be used a boundary for algorithms to avoid searching for triangles outside P. The convex hull is a subgraph of the Delaunay triangulion of P and is where most of the thinner triangles are located. This is due to the fact that the circumcircles for these edges often do expand outward from P as possibly very large circles. Figure 4.4 shows a few typical circumcircles on the convex hull.

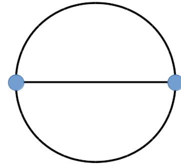

Figure 4.3: If two points in P can create an antipodal circle such as the one shown above. Where the circle contains no internal points from P, then it is a valid Delaunay edge.

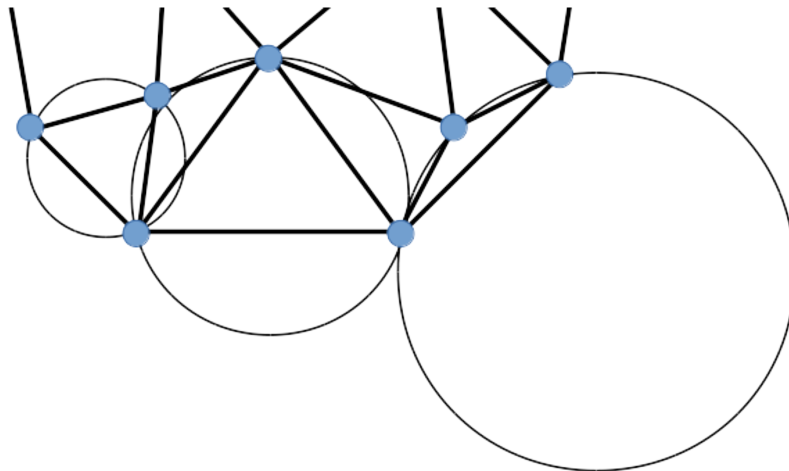

Figure 4.4: This diagram shows part of a triangulation and three circumcircles for Delaunay triangles where one edge from each lies on the convex hull. The triangles lying on the convex hull tend to be thin since their circumcircles are not limited by any points outside the hull.

4.2 Testing a Triangle for Validity

Determining whether a given triangles is a Delaunay triangle can be done in several ways. One of the simplest is to create the circumcircle of the triangle and check the distance from the center to points from P, if one is inside, it's not a Delaunay triangle. Another method uses the determinant of a circle based on three points in a particular order with a fourth as the variable to determine if a point is within the circumcircle created by the triangle of the first three. For integer values, both methods will yield exact and reliable results but in practice that many approximations must be made with floating point values where integer computations are not available due to the dimensions of some very large circumcircles. Figures 4.5 and 4.6 show

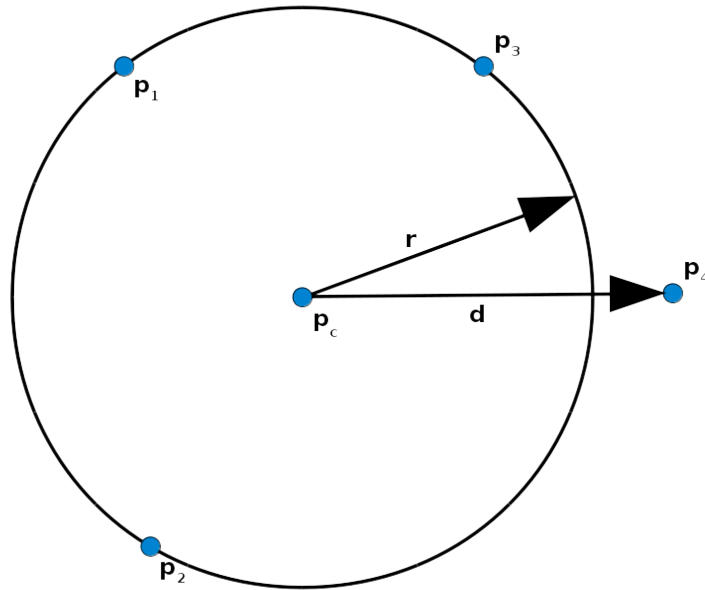

Figure 4.5: The points p_1, p_2 and p_3 form a triangle t that is being tested to see if it is a Delaunay triangle. The point p_c is the center of the circumcircle, p_4 is the point being tested against the circumcircle, r is the radius for the circle and d is the distance from p_c to p_4. As long as $d \geq r$ where $p_4 \in P$ then t is a Delaunay triangle.

how these techniques can be used.

4.3 Standard Search Triangulation

Standard Search triangulation very simply attempts to exhaustively create every triangle combination and uses the above tests to check if it's a Delaunay triangle. This is an $O(n^3)$ operation and is not recommended for use with all but the smallest values of n where it might be faster than creating the minimal support structures of other algorithms.

4.4 The Convex Hull

To calculate the convex hull the QuickHull algorithm is well-suited. It calculates the hull by inflating a minimal hull made of extreme points along an axis and recursively expanding each edge of the hull as it finds more distant points along that edge's orthogonal axis. QuickHull can make good use of

$$\begin{vmatrix} p_{1,x} - p_{4,x} & p_{1,y} - p_{4,y} & (p_{1,x} - p_{4,x})^2 + (p_{1,y} - p_{4,y})^2 \\ p_{2,x} - p_{4,x} & p_{2,y} - p_{4,y} & (p_{2,x} - p_{4,x})^2 + (p_{2,y} - p_{4,y})^2 \\ p_{3,x} - p_{4,x} & p_{3,y} - p_{4,y} & (p_{3,x} - p_{4,x})^2 + (p_{3,y} - p_{4,y})^2 \end{vmatrix}$$

Figure 4.6: This matrix takes takes 4 input points and the determinant describes if the fourth point lies inside, on or outside the circumcircle of the first three. The first three, p_1, p_2 and p_3 are given in counter-clockwise order and form a circumcircle, p_4 is the point being checked against. The determinant of this matrix is d. If $d \leq 0$ then p_4 lies on or outside the circumcircle.

the partitioning done on the points to limit how many points it searches to expand the hull and runs on average in $O(n\ log\ n)$ time.

4.5 Closest Neighbours of a Point

Using the closest neighbours property it is possible it is possible to find the Gabriel graph[2] of P, which accounts for about 70% of the edges in the Delaunay triangulation. But in practice searching exhaustively for all the edges is inefficient with regard to time spent per edge. The closest neighbours algorithm instead selects P_n, a subset of P containing at least a set number of points. The algorithm then tests these points for the closest neighbour property and stops after it finds one or more such neighbours. Some algorithms are dependent on each point having at least one neighbour to start with and so it finds at least the nearest neighbour each point in P.

4.6 Radial Triangulation

Radial triangulation is a triangulation completion algorithm and relies on both the convex hull being calculated as well as each internal point have at least one edge determined beforehand. The radial property is displayed in that each internal point has triangles that make a fan around it.

Starting from a point p_1 in P the algorithm finds an existing edge from p_1 to p_2. The dividing line l defined by p_1 and p_2 is then used to search P for points at a positive distance from l. These points are treated as candidates for p_3, the points found which create a triangle t defined by p_1, p_2 and p_3 are then compared until a t is found whose circumcircle's center is the closest in distance to l. This circumcircle can optionally be validated by using one of the above techniques and if it passes the circumcircle property the new edge from p_1 to p_3 is added to the triangulation. Point p_2 is then replaced by p_3

and the same search repeats until the next p_3 is the original p_2. Figure 4.7 illustrates this algorithm roughly.

For a point p_1 on the convex hull, the algorithm treats it as a special case when creating the fan, the original p_2 is set as the next point counter-clockwise around the hull and the end point for p_2 is set as the next point clockwise around the hull. The ensures that the fan being created is restricted internally to the hull and sidesteps the issues of having the algrithm search in vain for edges outside the convex hull.

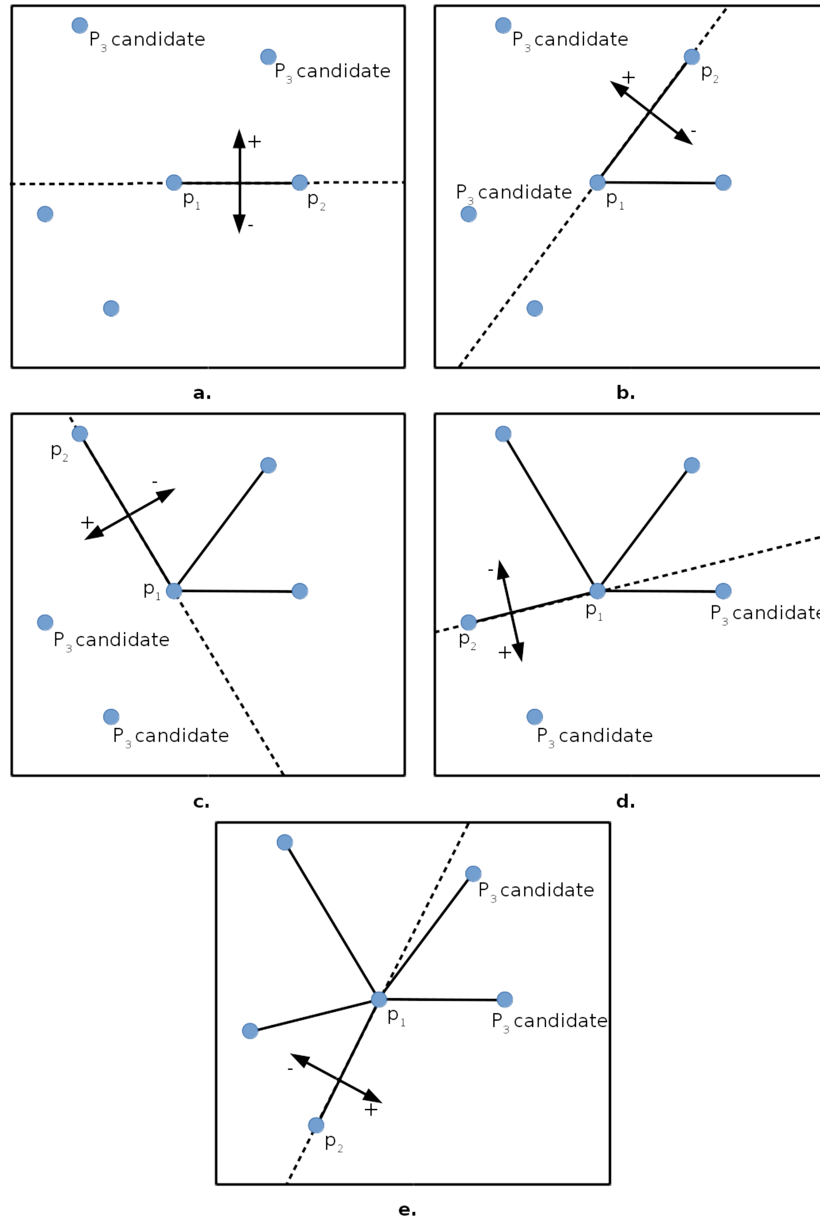

Figure 4.7: *a.* through *e.* show successive searches for points in a radial manner. This creates a "flower" of edges out from p_1. p_1 is the point being procesed by the algorithm, it finds p_2 as a starting edge and searches for p_3 candidates which can become the next p_2. Solid lines indicate edges that have been found, the dashed line is the line dividing the search space into positive and negative distance, the arrows indicate which directions are positive and negative. After *e.*, the search is back at its initial p_2 point and the search is complete.

Chapter 5

Data Structures

The choice of data structures to use during triangulation is critical with regard to performance. Choosing a complex data strcture can needlessly decrease performance with no gain otherwise. Likewise a naive data structure might forego performance optimizations that could be implemented with little extra complexity. The three main data structures that needed careful evaluation in this project were: the set of points and edges, the partitioning scheme for that set and the convex hull. The last of those became more obvious as a structure to evaluate once other optimizations were already in place.

5.1 The Set of Points

The first data structure to be concerned with is the main store of points, which includes their coordinates, the edges going between points and other data each point may have. The program produced here is meant to be scalable to any number of fields per point and should accommodate this with minimal performance and storage overhead. P will be regarded as a set of points regardless of what data structure lies beneath it. Each point in P gets a unique identifier, either an integer or a reference.

5.1.1 Point As a Class

The first and most simple solution is to create a class: *Point*. *Point* would contain all the components of a single point, the coordinates and edges most importantly. The edges are stored as a array of references to other instances of *Point* and are added to each *Point* as part of the triangulation process. See figure 5.1 for a diagram of this approach.

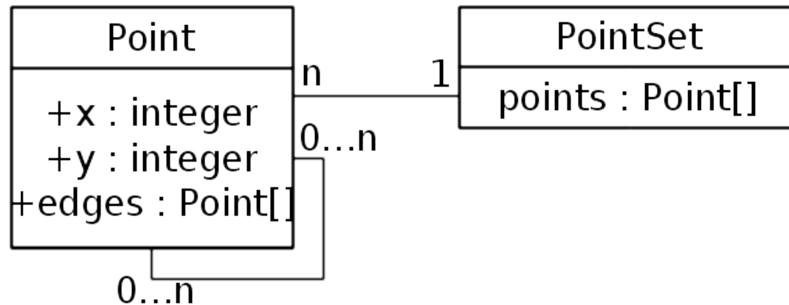

Figure 5.1: UML diagram of class *Point* and how it would relate to P represented as *PointSet*.

5.1.2 Interleaved Array of Components

Improving on the previous solution of using objects for each point, another solution that uses one large contiguous array of values to contain all the components(or indices to other arrays of component data). For example, given points with just X and Y coordinates this could be stored in an integer array containing $2n$ integers with every other integer being X and Y and each point being separated by a stride of 2 elements in the array. Figure 5.2 shows how such an array may be structured, with an additional example of differently typed components.

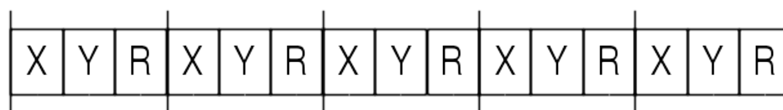

Figure 5.2: Diagram of an interleaved array. The points contain the integer components X and Y as well as the integer reference R as an index into another array of a different type component. The larger lines delimit each point in the array.

5.1.3 Seperate Component Arrays

Building on the use of arrays to store the component data, an approach can be devised in which each component is stored in a separate array. In this approach each point is identified by one index that it shares across all

component array e.g. the same index in both x and y coordinate arrays for the same point. This way each point is represented by a row crossing the columns of components. See figure 5.3 for an example of this structure.

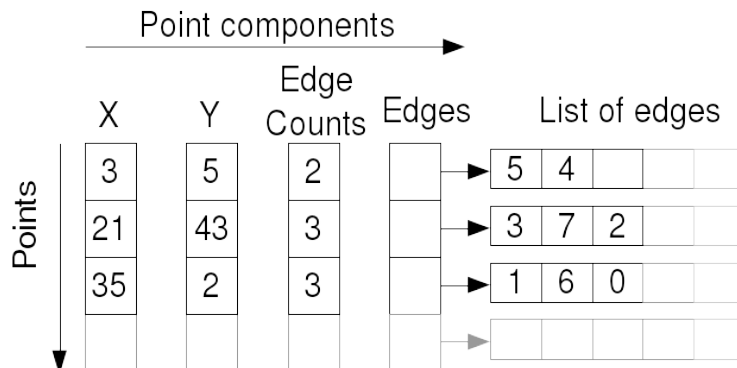

Figure 5.3: Diagram of a separate component array

5.2 Sorting and Partitioning the Set of Points

Simply having a large set of points isn't sufficient for efficient triangulation. More often than not, the edges between points that form the triangles will be between just the central point p_1 and its closest neighbouring points P_n. Consequently as n grows, the inefficiency of searching the entire set of points rapidly becomes a bottleneck in the process($O(n^3)$ for the most naive algorithms).

By selecting an appropriately sized P_n from P, the impact of having to search a larger P can be greatly minimized. It then becomes obvious that having some sort of spatial sorting to group nearby points together would be a great benefit to alowing algorithms to work with subsets of P rather than the entire set for each operation. By selecting the right kind of spatial sorting, a closeness property can be maintained where points close to each other in space are also close to each other in memory.

5.2.1 Sorting

Sorting P in some manner is the first and most obvious way to create assumptions about the order of the points in P. For example P could be

sorted along one of the axes, usually one of the elementary axes of the point components. Since 2D points are in question, the simplest way to sort them would be along the x-axis or y-axis, but sorting could be done along an arbitary axis. See figure 5.4 for simple illustrations. Other types of sorting can be done as well such as sorting in with regard to distance from a central point or sorting radially around a point, but these approaches will not be discussed here as they are not interesting when sorting P. However, one usage of sorting that will be discussed is the sorting of subsets of P in order to create better assumptions about the spatial locality of points. The section on hierarching segments expands on this.

The primary benefit of sorting along an axis is that when iterating over P or a subset of P, consecutive points in memory will also lie consectively along the axis that P has been sorted on.

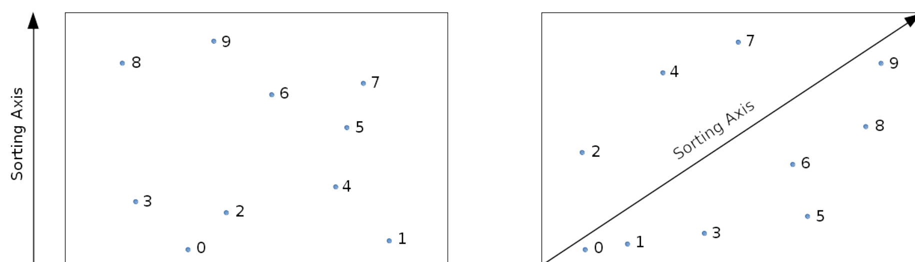

Figure 5.4: Diagram of sorting along one axis

5.3 Partitioning

Once the points are sorted, the next obvious optimization is to partition the sorted set P into groups of points. These groups would be ranges of P divided up either evenly or according to some other criteria. The purpose is to easily jump into a subset of P given some search criteria instead of iterating through all of P and finding the desired subset through naive search every time.

5.3.1 Segment Partitioning

The simplest way to do partitioning is to divide P into p equal-sized portions of P and record the starting index in P for each of them as well as some indication for how to select this portion relevant to how P is sorted. This could be for example recording the X or Y values of the first point in the

segment or other data that makes it easy to define this segment as relevant to a specific query for a subset of points. Figure 5.5 illustrates how sorted points with segments may be queried for a useful subset of points.

This approach works well to limit the number of points an algorithm must work with at a given time, but it can lead to non-local points being fetched if P is not at least near-uniformly distributed. Increasing the number of points per segment will on average lead to more uniformly-spaced segments and better locality when asking for the neighbours of a point but decreased performance as a larger subset of P will be returned.

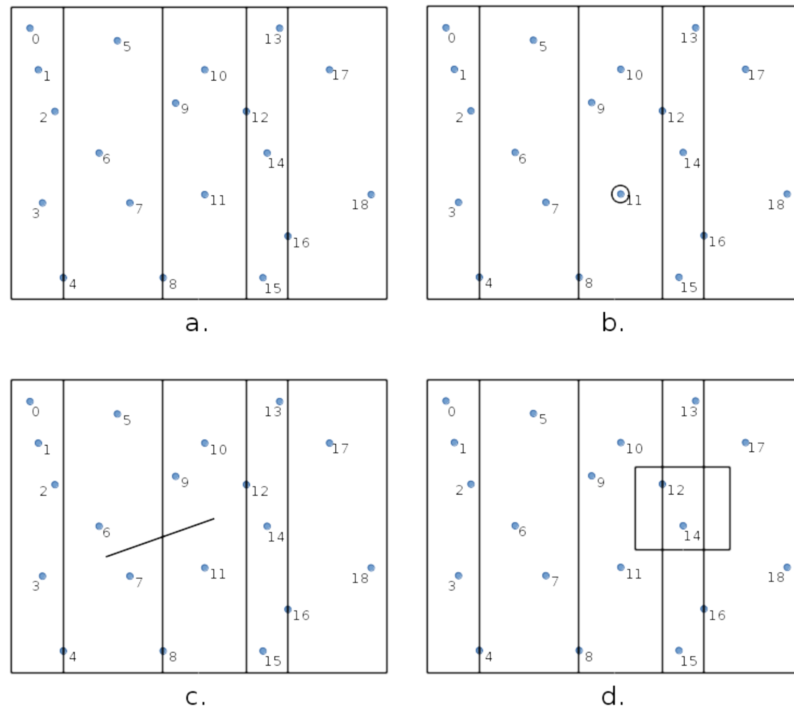

Figure 5.5: Visualizes a segmentation partitioning of P where each segment contains the same number of points. Segment lines show the start of a segment and intersect with the first point. *a*. A possible segmentation containing 4 points in each segment, with the points sorted along the x-axis. Except for the first segment, the start of each segment intersects with a point and extends up to but not including the point starting the next segment. *b*. The segment containing the circled point is fetched returning points from 8 to 11. *c*. The segment thats intersect with a line are fetched returning points 4 to 11. *d*. The segments containing the points in the box are fetched returning points 8 to 18.

5.3.2 Volume Partitioning

A different approach to fixed-size segments is fixed-volume segments, where each segment spans the points contained in a certain volume. Each segment is then of variable size but covers the same volume of space as other segments. This immediately leads to better locality for queries of segments as some guarantees can be made about the locality of the returned subset of points. Figure 5.6 shows an example of volume partitioning similar to segment partitioning in figure 5.5.

One disadvantage to this approach is that the density of segments may vary greatly, with some segments even empty. This can lead to wasted space spent representing the segments.

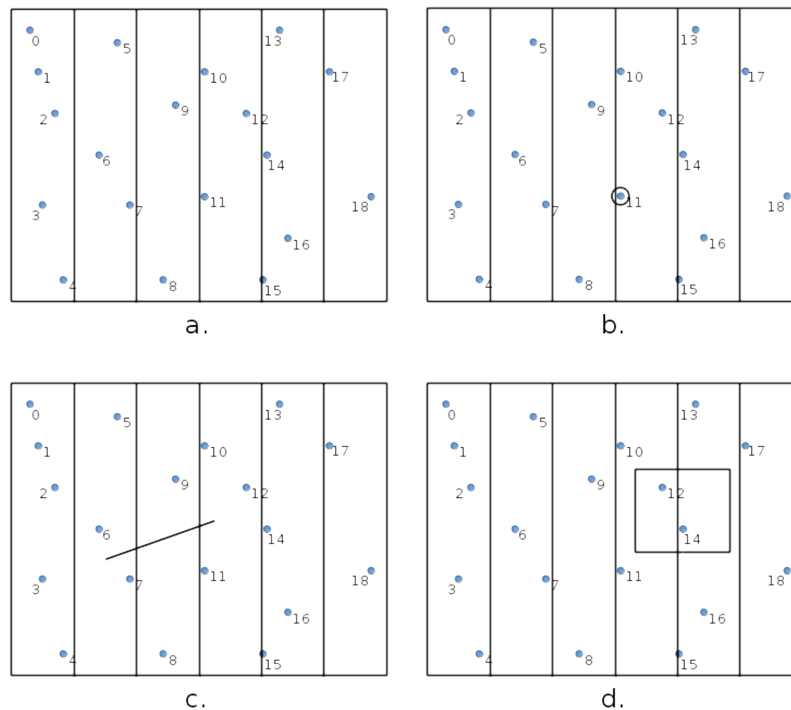

Figure 5.6: Visualizes a volume segment partitioning of P where each segment contains the same volume but a variable number of points. Segment lines show the start of a segment and if they intersect with a point it is included in the segment extending to the right. *a.* A possible volume segmentation containing 6 equally-spaced segments along the x-axis. *b.* The segment containing the circled point is fetched, returning points from 10 to 12. *c.* The segments that intersect with a line are fetched returning points 6 to 12. *d.* The segments containing the points in the box are fetched returning points 10 to 16.

5.3.3 Hierarchical Segment Partitioning

The previous approach, while efficient for partitioning points, will eventually lead to problems with the number of points in each segment and where they are in relation to each other. As n grows, if the number of segments does not grow with it, the number of points per segment will increase and lead to decreased performance when each returned subset of a query is much larger than it needs to be. If the number of segments increases proportional to the number of points, the segments will become spatially thin and the points contained inside will be less spatially related on average.

It is desirable that segments don't have dimensions that are too dissimilar and that each one has a manageable number of points contained inside, the solution is then to create a hierarchy where larger segments contain subsegments that try to maintain similar dimensions. One immediate advantage is that the segment a point lies in and the adjacent segments in every direction will have points that are closely related and thus enhance the locality of queries for points.

One drawback to this however is the increased cost of creation and the cost of addressing these segments, it is a necessarily more complex data structure. This approach can also be done with subsegments described as above, either containing a fixed number of points or containing a fixed volume. If the subsegments contain fixed numbers of points this can easily lead to adjacent subsegments being unintuitive to address in queries for subsets of P. Figure 5.7 shows an example of this in practice.

As the number of levels in the hierarchy increases, this problem persists and creates a great of complexity in getting a subset of P that matches a query.

5.3.4 Grid Partitioning

Addressing the issue of mis-aligned segments hierarchical model, a flatter, more regular hierarchy can be created. This partitioning model is more like a grid. It is based on volume partitioning the points to the number of dimensions they have. N-dimensional points would get an N-dimensional grid and so on. In the case of the 2-dimensional points in P, either one of the X or Y axis could act as the first dimension and then each of those segments would be divided into segments of the other dimension respectively. The smaller segments are much like cells in a grid.

The size of cells can easily be tuned to the right size by altering the number of segments in each dimension to reach an appropriate average number

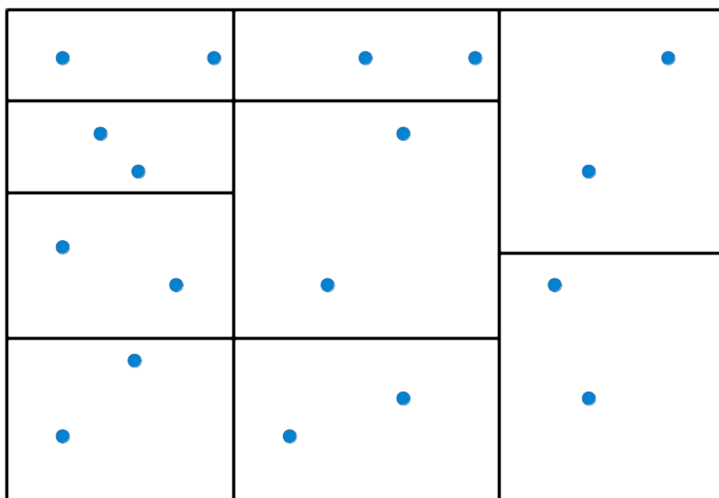

Figure 5.7: Shows a typical issue with addressing fixed size subsegments. Adjacent segments may have differing numbers of subsegments and those may not be directly adjacent to each other. This means that finding the right subsegments to return for a query may require some significant searching.

of points in each cell, C_a. Since the hierarchy has very few levels(N levels for N-dimensional points) and since the number of segments in each dimension can be pre-determined, the lookup for each cell can be reduced to a lookup in an N-dimensional array.

When it comes to addressing larger subsets than a single cell the regularity of the grid can be exploited. Since the grid is entirely regular in the size and shape of its cells, a box can be made and imposed on the grid. The corners of this box intersect with the boundary cells that need to be recorded as delimiters for the subset. Since the Nth dimension of segments are contiguous and match up across rows(can be regarded as columns across the grid), the resulting box can then consist of a series of 1-dimensional rows or strips running along the Nth dimension in the grid. This may not be as efficient as returning a pre-existing subset from a deeper hierarchy that divides on more than just the elementary axes, but depending on the size of the cells it can return a much smaller subset that more closely matches the desired one without sacrificing too much performance.

The time to access any cell in the grid is $O(1)$ due to the strict regular structure of it. The time to create a subset of the grid can vary depending the dimensions of the query and how many cells need to be returned.

The biggest problem with the grid partitioning is that as the number of

cells increases or C_a decreases, the amount of memory required to store it increases. This is a parameter that must be tuned appropriately to avoid having the grid consume too much available memory. The creation of a grid with many small cells is also a potential issue as each cell must be given a starting index for which points it holds. C_a can be anywhere from close to 0 and up to n so care must be taken to set it appropriately for the right space vs. performance trade-off.

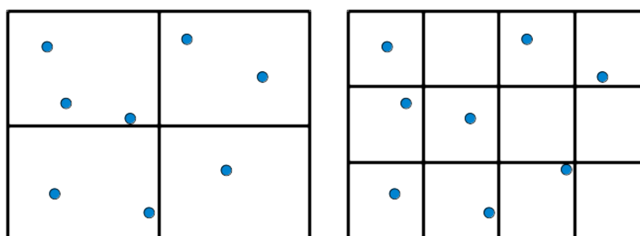

Figure 5.8: If grid cell size is too small, space will be wasted representing empty cells as shown here. Tuning the grid size is therefore a trade-off between resolution of point addressing and required space to represent the grid in its entirety.

5.4 The Convex Hull

One of the first steps in generating the Delaunay triangulation is to calculate the convex hull C. C is the volume constructed from the points in P and is defined by the subset P_c and the edges between them E_c. E_c forms a closed, convex hull where each e_c in E_c is defined by two points in P_c. E_c are created such that the points P/P_c are all contained inside C or intersect with E_c. E_c may have parallel edges, and if it does then all intersecting points are part of P_c. The convex hull is then the smallest set of edges, E_c, defined by points in P and described by the subset P_c that contains or intersects with P.

The convex hull is a subgraph of the Delaunay triangulation and as such each of the edges on the convex hull are all edges of Delaunay triangles. For many algorithms they are used as the starting points to recursively triangulate the remaining internal points.

Looking aside from the algorithms used to generate it, the convex hull needs to be stored in a data structure that can be efficiently queried. Algorithms often need to know if a point is part of the convex hull and which points are its neighbours in the convex hull. The data structre must therefore have quick read accesses and must be iterable.

Having no data structure to store the hull but rather making checks to determine if a point was part of the convex hull can quickly become a very expensive operation as n grows. Many calculations would need to be performed every time a point was queried so it was determined that an additional data structure to manage the convex hull was necessary.

5.4.1 Querying the Hull for Points

The primary purpose of having a persistant data structure for the convex hull is to query whether a point is part of it. One could solve this by having an extra boolean component per point that would indicate whether it was part of the hull or keep a list of points to indicate which points are on the convex hull. These points, P_c, typically number about $\sqrt{n}$ and as such are usually just a small subset of P.

A list of some sort to contain the hull is therefore a practical solution. It can be queried for the existence of a point and return the answer. But this solution would still suffer as n increases. A typical P_c for 10 million points numbers around some 8000 points. Instead of searching through such a list every time, it's better to keep a hash table of the points and query it when needed.

5.4.2 Traversal of the Hull

In addition to querying the hull about whether a point is part of it, some algorithms can also benefit from knowing which of a point's neighbours are part of the hull. For any point on the hull, there will always be two neighbours also on the hull(as long as $n > 2$). If there references to these two neighbours are also stored, it's possible to traverse entire hull for the benefit of algorithms that may need it.

By using a double linked-list structure, the points of the hull can be linked together and traversed in either direction. If this list is kept in addition to the hash table of points, the convex hull can be both queried and traversed efficiently.

Chapter 6

Implementation

The program implementing and testing all of the above data structures and algorithms is written in Java. It can be configured to run each of the different algorithms for triangulation and make use of different data structures as backing for them. The implementation has been designed in fairly procedural and modular manner to make future extension with new data structures and algorithms a feasible task.

When the triangulation algorithms have run their course it is possible to output the resulting data either to file or to output it graphically in a simple window which shows the points, their identifiers in the set of points and the edges between them. The graphical representation is not exactly accurate since it becomes scaled down to fit a window for display.

6.1 Data Structures

The various data structures have been implemented in a fairly straightforward manner. Several of them contain optimizations such as precomputed values where appropriate or public access to members to assist in ease of use. The data structures includeded in the final program are the separate arrays per component, the grid partitioning scheme and the convex hull. Following is a detailed description of the design and implementation of these data structures.

6.1.1 The Set of Points as Component Arrays

The set of points as component arrays are implemented in the *PointSet* class. An UML diagram in figure 6.1 shows an overview of the class members and methods. The class is implemented in very straightforward manner, nearly

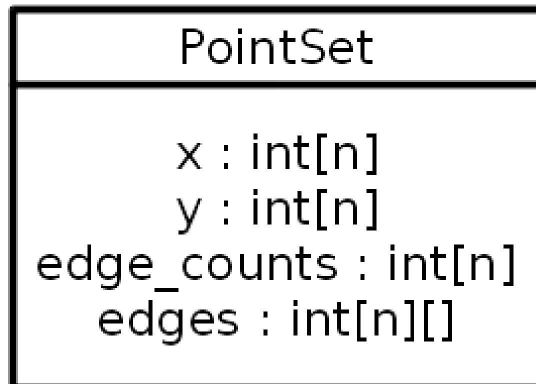

Figure 6.1: The *PointSet* when implemented with seperate component arrays has four members of size n, where n is the number of points in P. The four members are arrays for: the x and y coordinates, the edge counts for the number edges found on each point and finally the array of arrays that contains the edges found for each point.

as a pure data store with thin wrappers for querying the components of each point.

The components used for points in the default program are integers for the x and y components, and an array of integers for the list of edges each point has as well as an integer indicating the number of edges already in the array. As described earlier it is trivial to extend the number of components to include other attributes such for example a boolean array to indicate special properties for each edge.

The x and y components are considered immutable after the initial sorting and partitioning and should not be altered despite public access to their arrays. When adding an edge, the edge counter for that point is incremented and checked against the bounds of the edge array, if the array lacks space for another edge, a larger one is allocated and the existing edges are copied over. The default policy for increasing the array size is to double it.

A copying member method is supplied to allow copying a point with all components from one set to another. If additional components are added to the set of points, this method will need to be modified or overriden as well.

6.1.2 The Grid Partitioner and Sorting

SearchGrid is the class which implements the grid partitioning scheme and contains the metadata about *PointSet* to assist viewing *PointSet* in a spatially sorted manner. The UML diagram in figure 6.2 shows the layout of

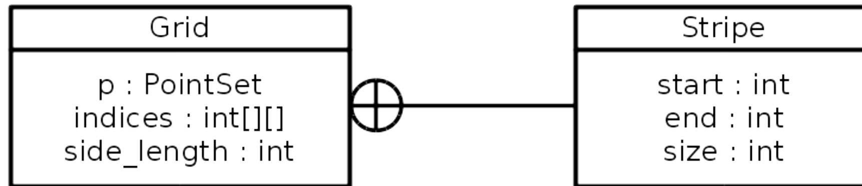

Figure 6.2: Diagram showing the two classes *Grid* and *Stripe*, where Stripe is an inner class instantiated by Grid and returned as results for a query. *Grid* contains a reference to P, the indices calculated to represent the grid and the length of one side a cell in the grid. *Stripe* represents a subset of a row in *Grid* and uses the start and end values for the cells it covers as its start and end members. The size member in *Stripe* is an optimization that is the pre-calculated difference between start and end.

this class. The inner class called *Stripe* is used to represent one subset of a row in the grid being being presented for processing.

The *indices* and *side_length* members of the *SearchGrid* are the main points of interest. The *indices* member contains increasing offsets into the *PointSet* describing where each cell of the grid begins and the size of each cell limited by the start of the next cell. The *side_length* member is used to transform from point-space coordinates to cell-space coordinates and vice versa.

The constructor for the *SearchGrid* takes care of creating the *indices* member. This is the most time-consuming part of the SearchGrid. This process is best described by the following algorithm and the diagram in figure 6.3. It boils down to sorting the *PointSet* on the y-axis, partitioning it into rows, sorting each row on the x-axis and then partitioning those as well. The offsets for each column are then recorded in *indices* and the grid is completely constructed. The $side_length$ member is calculated as part of the partitioning process, see the algorithm below and figure 6.3 for how it's determined and used here.

An important part of the *SearchGrid* is the capability to make subsets of the *PointSet* available. This is implemented as a set of methods in the *SearchGrid* that can return arrays of *Stripe* objects. The *Stripe* objects are as earlier described the subset of one row in the grid, several of these rows are combined in an array to form usually a rectangular section of the grid, but future implementations could use different size rows to create approximate other shapes such as triangles, circles and lines.

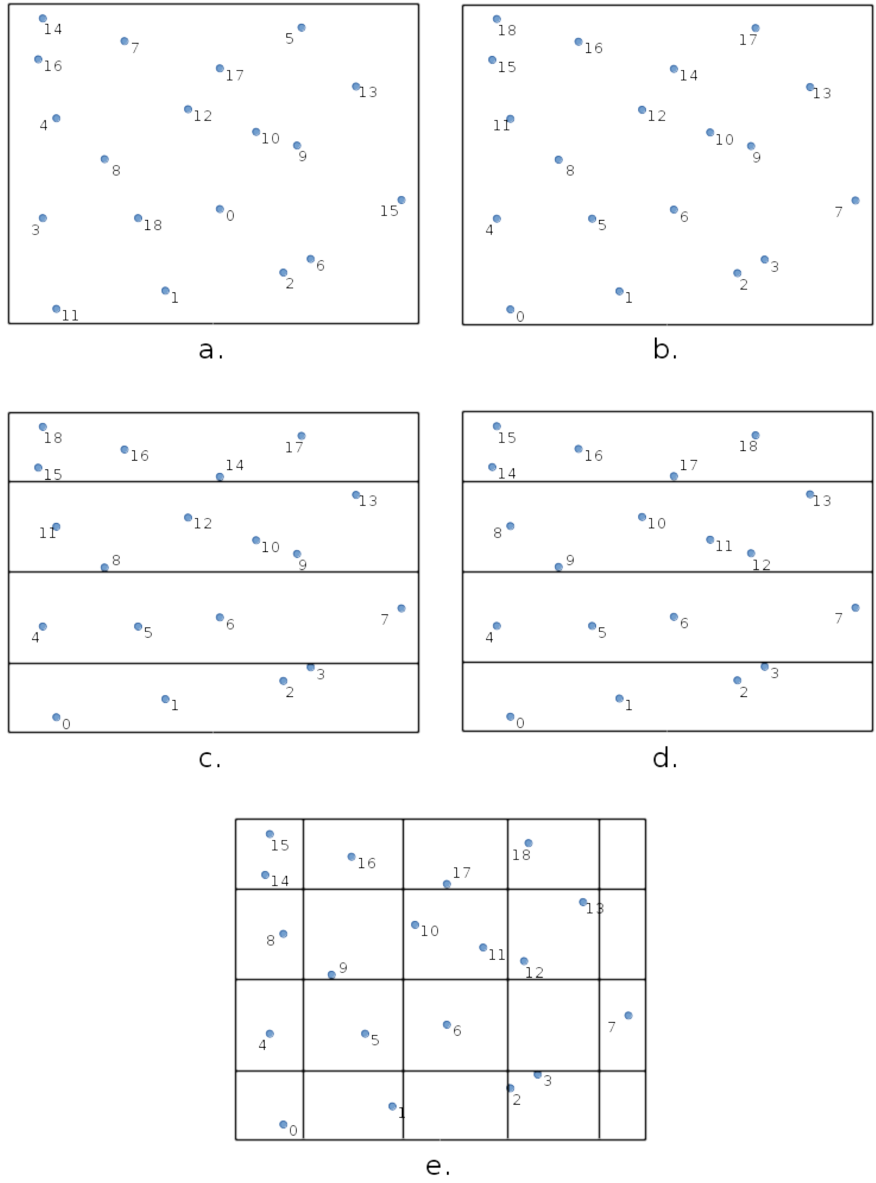

Figure 6.3: *a*. The unsorted points. *b*. P is sorted on the y-axis. *c*. P is partition on the y-axis, the lines indicate thew rows of the grid. *d*. Each row is sorted on the x-axis, note the top rows for examples of change. *e*. Each row is partitioned into equal columns to complete the grid.

6.1.3 The Convex Hull

The $ConvexHull$ class contains two simple data structures, a $LinkedList < Integer >$ member named $hull$ to contain the list of points on the convex hull in clockwise order and a $HashSet < Integer >$ named $hull_set$ which is used for querying the convex hull to find out if it a point is in the list. The $hull$ member is used to find the "next" and "previous" points on the hull for a given point by using the corresponding member methods. The $hull_set$ member is an optimization that's used to find the existence of a point on the hull instead of searching through the $hull$ member to find it.

6.2 Algorithms

Most of the algorithms implemented in the program are written in a straightforward manner. They often have some simple optimizations to limit the scope of searches but otherwise use no other smart tricks(such as exploiting the rest of the Gabriel Graph[2], creating a draft triangulation that can be refined and so on). The triangulation algorithms implemented are both a standard search algorithm and radial search algorithm as these two lends themselves most easily to parallelization in great part due to focusing on one point at a time.

The algorithms implemented in the program use the same technique to ensure thread-safety for the most part. When an edge is added from p_1 to p_2, only the edge from p_1 to p_2 is added, there are two reasons for this. The first is that the edge p_2 to p_1 will be found later anyway when p_2 is searched for edges. The second reason is that p_2 may currently be undergoing processing by another thread and the edge list for p_2 must be regarded as volatile as such, any concurrent modification has a chance of side-effects such as missing edges or invalid edge counts.

6.2.1 Nearest Neighbours by Standard Search

The standard search version of the closest neighbour search starts with a point p_1 and gets a box B from the search grid with at least $num_neighbours$ points centered on p_1. The algorithm then searches B for points named p_2 within a search circle C extending to the nearest edge. C is used as a limit for the distance to p_2 as it guarantees that the nearest neigbour property test will disgregard any points which may have hidden points outside B. Figure 6.4 shows how this may happen if C is not used to limit the search within B.

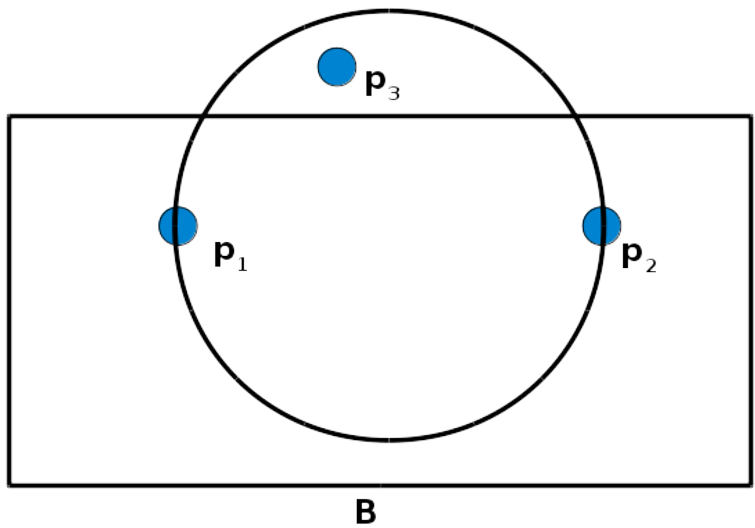

Figure 6.4: This diagram illustrates a problem with the nearest neighbour search. p_1 and p_2 appear to pass the near-neighbour property check, but since their antipodal circle passes outside B a point p_3 may still interfere without being detected when checking for collisions inside B. The solution is then either to use a large enough B or to restrict which points inside B can be checked to only those whose antipodal circle with p_1 would not pass outside B.

When an appropriate p_2 is found inside C, it is tested for the nearest neighbour property against p_1. If the property holds, the edge between p_1 and p_2 is added to the *PointSet*. If no point in B is found to pass the tests, a new B is created with a larger C and the search is run again.

6.2.2 Nearest Neighbours by Distance Sorting

Very similar to the standard search version of the closest neighbour search, the distance sorted variant creates a box B that contains at least $num_neighbour$ neighbours and contains the search circle C centered on a point p_1. Different from the standard search however is that B is copied into an array and sorted by increasing distance from p_1. Then, starting from the beginning of the list, each point, named p_2, is tested for both being inside C and against p_1 for the nearest neighbour property. For each point that fulfills both tests, the edge p_1 to p_2 is added to edge list for p_1.

Again different from the standard search is that due to the sorting by distance, once a p_2 is found that fails the distance test, the remaining points will also fail the test and the search can be stopped. Once again, if no edges have been found, a larger B for a larger C is created and the search is run again.

6.2.3 The Convex Hull by Quickhull

The construction of the convex hull takes place in the constructor for the *ConvexHull* class and uses the QuickHull algorithm to expand the hull from up to 4 initial points. The initial points are found by searching inward from the four edges of the grid, looking for the minimum x-coordinate, maximum y-coordinate, maximum x-coordinate and minimum y-coordinate. They are kept in this order specifically because the line function going clockwise around the hull will find distant points at positive distance from the intermediary hull edges. See figure 6.5 which shows how the line function direction afects positive distance.

If any points are found multiple times as part of the hull, they will only be added once. This accommodates for starting hulls that may contain three or even two points as the initial hull. For each edge E_c in the current hull, a box B is created to search for the most positively distant point from that edge. The search for edges inside each edge continues recursively until no new edges are found.

B is created to contain the start, p_1, and end, p_2, of the current edge. This subset from P can be chosen because any point that falls outside B

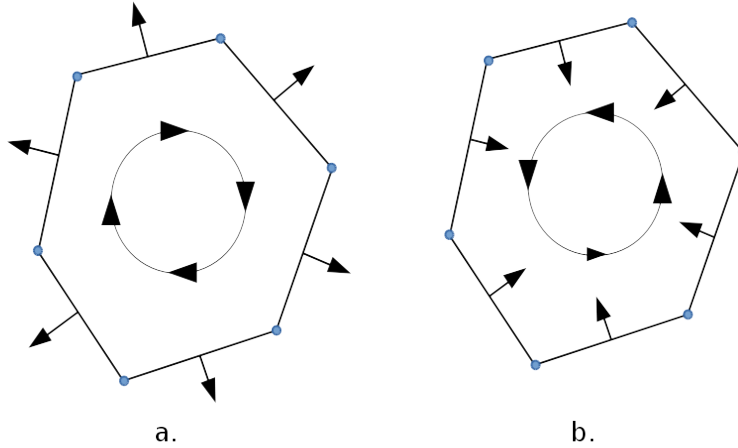

Figure 6.5: *a.* Shows the direction of positive distance from the edges if the line functions for the edges are created in clockwise order around the hull. Positive distance points out from the hull. *b.* Shows the direction of positive distance from the edges if the lines functions for the edges are created in counter-clockwise order around the hull. Positive distance points inward into the hull.

would violate the convex property of hull and be found by an earlier search for edges. Figure 6.6 illustrates how this works for an example hull.

To find the most distant point from an edge, the line function for that edge is created and the distance is calculated to each point tested. The distance calculated is the squared distance as taking the square root and finding the exact distance is unnecessary since the greater/less than relationship between the distances checked will remain the same when squared.

The algorithm runs through all the points in B and records successively more distant points until it exhausts B. If a distant point p_d has been found, two new edges from p_1 to p_d and p_d to p_2 are added to the convex hull. E_c then no longer exists as p_d has been inserted between the two in the *hull* member in the $ConvexHull$ object.

If the most distant point found lies on E_c, then p_d is checked to make sure it lies between p_1 and p_2, not outside them. If this holds, E_c is divided into two new edges as normal and p_d is inserted between p_1 and p_2. If multiple points are found on E_c, the one closest to p_1 is kept as p_d.

Figure 6.7 illustrates how the algorithm selects p_d from the available points around each E_c of the existing hull.

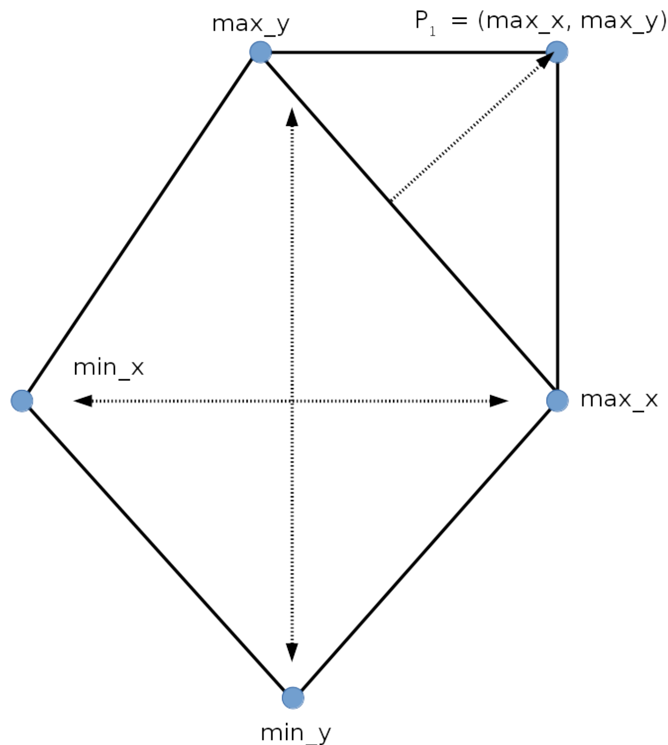

Figure 6.6: Point p_1 is the most distant point that can be found during a search outward from the edge between the points max_y and max_x. Any point more distant would replace either max_x or max_y in earlier searches. The triangle covered by max_x, max_y and p_1 will therefore contain any eventual point from P to create a new edge on the search for the convex hull.

6.2.4 Triangulation by Standard Search Radial Triangulation

This algorithm completes the triangulation of P only if every point p_1 has at least one existing edge to a point p_2. This means that it assumes a closest neighbour algorithm has previously been run to find at least one neighbouring point. Starting from the first edge for p_1 to p_2, E, a box B is created around E as a starting point for the search. All the points in B are tested as candidates for p_3. As is indicated in the algorithm's name, the triangulation is radial. In this case it means that any p_3 is always a positive distance from E, any p_3 at negative distance to E is disregarded. Secondly, for p_3 to be accepted, the triangle p_1, p_2, p_3 must be a Delaunay triangle. This algorithm draws inspiration from the work of Arne Maus[3].

To validate p_1, p_2, p_3 as a Delaunay triangle, the circumcircle, C_t, for

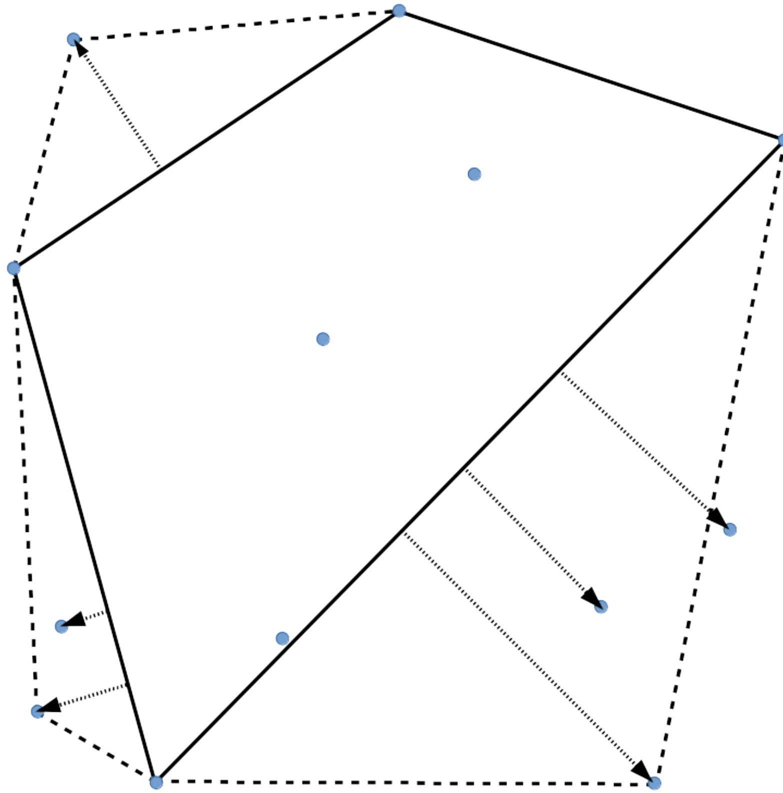

Figure 6.7: The dashed lines show new edges found by searching for distant points outward from each of the old, solid edges. Note that the most distant point from each edge can still fall outside the hull, it will be found on subsequent searches to expand the hull.

this triangle is created and a box, B_t, is created to encompass the subset of P in its neighbourhood. Each point p_4 in B_t is tested against C_t using the determinant check to determine if p_4 is inside C_t. If no p_4 is inside C_t, the edge from p_1 to p_3 is added. If no p_3 is found in B that creates a Delaunay triangle, then a bigger B is created and the search starts over.

The algorithm then assigns p_3 to p_2 and the edge E between p_1 and the new p_2 becomes the starting point for the next run of the loop in the algorithm. This loop continues until the original p_2 is once again p_2 and the radial search is complete. There is a special case for the convex hull in that the start and end points for p_2 are different. When it is found that p_1 is part of the convex hull, p_2 is set to the the next point counterclockwise around the hull and the end of the search is set to the next point clockwise around the hull. This way the radial search on the convex hull searches internally

in the hull for new edges. Once the end point for p_2 is reached for p_1 on the convex hull, the loop in the algorithm terminates as normal.

6.2.5 Triangulation by Nearest Circumcenter Radial Triangulation

Very similar to the previously described algorithm with the except that when B is searched for a candidate p_3, the circumcircle C_t is created for the triangle p_1, p_2, p_3 as before. In this algorithm however, the p_3 considered best candidate is the one with the circumcenter for C_t nearest to E. After searching through B for these circumcircles, the best one found is tested with the determinant test against possible points p_4 as described above. If the candidate for p_3 did not create a Delaunay triangle, the search range is expanded, a new B is created and a new candidate for p_3 is searched for, this continues until p_3 has been found.

6.2.6 Triangulation by Existing Edges Radial Triangulation

This algorithm is a smarter version of the nearest circumcenter search. Instead of trying all nearby points, it uses existing edges as a starting point to narrow the search. Any existing edge found to be on the right side of E is used to create a subset of P that must contain the next p_3. If no such existing edge is found, the same search described above is used to find p_3.

6.3 Thread Labour Division

The number of threads spawned for the triangulation is simply the number of cores reported by the JVM during runtime. These threads have the labour divided between them simply by being given a range of points to work on. The division is done as equally as possible since the expected input points are uniformly random and therefore the amount of work per point is expected to approach an equal average across all the threads. This division also works well with the program since it iterates over points in a given range and calculates the edges for each.

Chapter 7

Results

7.1 Data Structures

7.1.1 Data Types, *Integer* vs *int*

Early on during development the decision was made to avoid generic types in Java such as *Integer* in favor of more primitive *int* data types. *Integer* objects need to be allocated with new, even as part of arrays where they should ideally be packed in memory. Some number for the use of *Integer* vs *int* are show in figure 7.1. While the triangulation itself only gains a 1.25x speedup, the construction of the *Grid* gains a much more substantial speedup at 5.58x. This difference is primarily due to the sorting algorithm taking significantly longer to *Integer* objects than *int* primitives(Not shown in the table, 15240ms vs 2300ms). Finally, the use of *Integer* instead of *int* lead to almost exactly 2x consumption of the heap after triangulation was complete.

Data Type	*Integer*	*int*
Construction and Population of *PointSet*	6731ms	2698ms
Construction of *Grid*	20834ms	3732ms
Triangulation Time	13775ms	10984ms
Time Spent Triangulating Per Point	10.759859μs	8.670928μs
Heap used at the end of triangulation + GC	1502MiB	753MiB

Figure 7.1: Some of the relevant numbers for *Integer* vs *int* and how much time they take in the run-time and memory consumption of the program. The numbers are taken from the triangulation of 10 million random points using seed 0. Data structures used are the seperate component array and grid partitioning while the triangulation algorithm run is the radial triangulation using nearest circumcenter search run on 8 threads.

7.1.2 Point As a Class

Despite being attractive in an organizational manner, two performance problems are immediately evident with this approach.

The first of these problems is the indirection involved in accessing the data of each *Point*. Because each *Point* instance is a reference type in Java, they may lie completely at random in memory, making caching unpredictable and at worst only the data for the point being accessed is cached by the CPU.

The second problem is the memory overhead requirement for each object in Java. Each instance of *Point* has a fixed overhead that becomes a limiting factor as the number of points grows(Typical overhead for a HotSpot JVM is between 4 and 8 byte plus the other fields in the object).

7.1.3 Interleaved Array of Components

This approach has a few advantages, the first being that the overhead of object storage is eliminated with the exception of the overhead for the entire array. Another advantage is that the point components are laid out contiguously in memory, this is favorable to assist the CPU in caching entire points or even several points at a time when accessing the store. There are however some problems with this approach that must be addressed when adding new components to the points and the overhead involved.

The first of these problems is the complexity of addressing components of the points. Arithmetic must be done upon access to find the correct index for the component of the point being addressed. Additional complexity is involved in addressing non-integer data where these components would need to be kept in seperate arrays and indices to those arrays be kept as components to each point in the primary array of points.

This extra indirection leads to the second performance problem. The cost of the extra arithmetic and extra memory reads from possibly disparate locations for each access as well as the increased storage requirements in cache per point all negatively impact the performance of this approach and limit the number of points the CPU could keep in cache at a given time.

Finally, since all the components or their indices lie contiguously in memory, they will be cached by the CPU whether or not those components are ever in use inside a given algorithm. This in turn means that the effective payload per memory read is reduced. Depending on the number of components per point, maybe only a single point's components will be cached.

7.1.4 Seperate Component Arrays

The memory overhead is lower compared to the interleaved array as indices into each array are the same across the entire point and don't need to be stored as their own component.

The caching behaviour of this approach is also favorable as a memory read of a component will only cache the same component of other points, ensuring that the amount of useful data being cached is kept at a much higher level than the interleaved array. Since each component is kept in its own array, this means those components can be cached separately upon being read from memory. This can often lead to twice as much useful data being cached compared to the interleaved array where probably every component after the one being read is cached regardless of usefulness.

No significant disadvantages to this approach have been identified.

7.1.5 Caching Behaviour in PointSet Alternatives

In addition to their other differences, the alternatives for *PointSet* also have different behaviour with regard to how the CPU tries to cache them.

The alternative with each point as an instance of *Point* comes out worst, due first and foremost to the fact that every instance must be assumed to live in disparate memory sections, not sharing any cache lines. Due to this, the caching efficiency is very low for these objects with at most a single entire point being cached on read. Typical Java overhead for an object is 8 bytes fixed overhead for *Object* instances plus the components of the point. Each point with an x, y and z coordinate will then use 20 bytes for data and because of memory alignment be padded to 24 bytes. It is possible that other *Point* instances can lie directly after each other in memory, but Java makes no such guarantees about allocation of objects.

The other two approaches using arrays have earlier been described with regard to how the CPU can cache them. See figure 7.2 for a simple view of how the CPU can cache the different approaches.

7.2 Algorithms

7.2.1 The Convex Hull by QuickHull

The QuickHull algorithm works efficiently and runs in negligable time compared to the rest of the program. It runs in $O(n \ log \ n)$ time and completes at around 300ms for $n = 10 million$ on the test machine. It exploits the grid

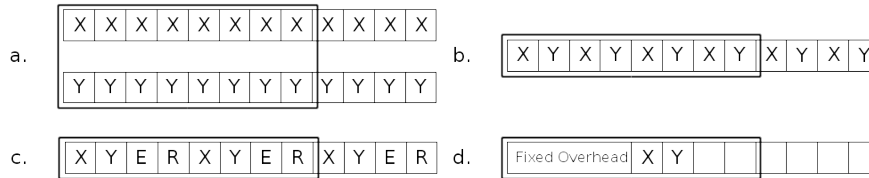

Figure 7.2: Each X or Y coordinate is an *int* composed of 4 bytes. The example shows the caching behaviour of an Intel Core i7 which uses 64-byte cache lines. In both examples the X and Y coordinates of first point from the left are read. The black boxes show the data being cached. Example *a.* shows the behaviour of seperate component arrays. The X and Y coordinates are in separate arrays and a total of 8 pairs of elements are read and stored in cache. Example *b.* shows the interleaved array where X and Y coordinates are interleaved in memory and stored in one array. A total of 8 elements are read in to cache but only 4 pairs of elements. Example *c.* shows the interleaved array with more components per point and demonstrates how the caching of useful data goes down as the number of components increases. Example *d.* shows the caching behavior when using a class reference for each point and its components. Just the single point can be regarded as being cached on any given access.

to create subsets effectively and searches increasingly smaller subsets as it refines the edges of the hull. Caching on this algorithm is most useful in that points are prefetched for the larger search spaces in the early edge searches while later on in the edge refinement process the cache will keep close to the entire search space in cache and be able to quickly iterate over the points being tested.

7.2.2 Closest Neighbours

Standard Search Closest Neighbours

This simple algorithm is the algorithm for finding the closest neighbours which works most efficiently when the JIT-compiler is enabled. It runs in $O(n)$ time since it attempts to always search the same number of points(*num_neighbours* is the target number of points) and checks these for the closest neighbour property. Due to the points always being searched in order of memory and tested in order of memory, it is believed that caching benefits this algorithm greatly as well the as the simplicity lending itself to easy optimization by the JIT-compiler.

Distance sorted Closest Neighbours

This smarter version of the closet neighbours algorithm is better in theory than the standard closest neighbours search but it is believed that the implementation of it is not easily optimizable by the JIT-compiler. The run-time is thus higher than the standard search despite the smaller search space possible through the distance sorting. The distance sorting itself may have an impact on the algorithm for sorting the points is not an efficient one(selection sort has been used).

7.2.3 Triangulation

Standard Search Triangulation

The standard search for triangles progresses in the most linear fashion possible and always iterates over points as they appear in memory. This leads to the algorithm being deceptively fast due to the CPU-caching being exploited. However, this algorithm makes no effort to use any of the "smart" shortcuts that are standard in the other algoritms, such as defining special cases for the convex hull, stopping when all edges have been found for a point or limiting the scope of the search for points to any particular spaces. The algorithm itself is runs in $O(n^3)$ time and for small numbers of points this is acceptable as the caching offsets the lack of efforts to limit the search space.

Radial Standard Search Triangulation

The standard radial search is an improvement over the standard search primarily in that it has a stop condition for the search outside of simply exhausting the search space. Additionally it limits the scope of the search to the neigbbourhood of p_1 by using the *Grid*, only searches around a point in the same direction and has special cases for points in the convex hull. These technqiues combined make it the best algorithm implemented in the program so far. Running in $O(k^2)$ time, where k is the number of points it typically has to search through in the boxes it creates for each point.

Radial Search with Nearest Circumcenter Triangulation

This approach is a slight improvement over the standard radial search in that it only attempts to validate the point in the box found which is the most likely to be p_3. Since it runs a simple circumcenter calculation on the rest of the points, it ends up running in $O(k^2)$ time just like the standard radial search, but where k is on average smaller than the standard radial search. Both this and the standard radial search make good use of CPU-caching

by searching iteratively through the boxes returned and reusing them where possible.

Existing Neighbours Radial Triangulation

This algorithm is technically superior to both the standard radial searches as it can use them as a fall-back to the method of using existing neighbours as a base for the points. However, this algorithm performs more poorly than the standard radial search since it searches much more randomly than the standard radial search. The neighbours it finds can often be at such angles that the circumcircle defining the search space is very large compared to the search space that typically develops in the standard radial search. The algorithm runs in $O(k^2)$ just like the other radial searches, but with a larger value for k.

7.2.4 Algorithm Performance

The triangulation algorithms scale similarly in a multi-threaded environment. Figure 7.3 shows the speedup for different numbers of cores. The relative speedup decreases as the number of cores increases, most likely because the memory bus becomes saturated with requests and can't keep up.

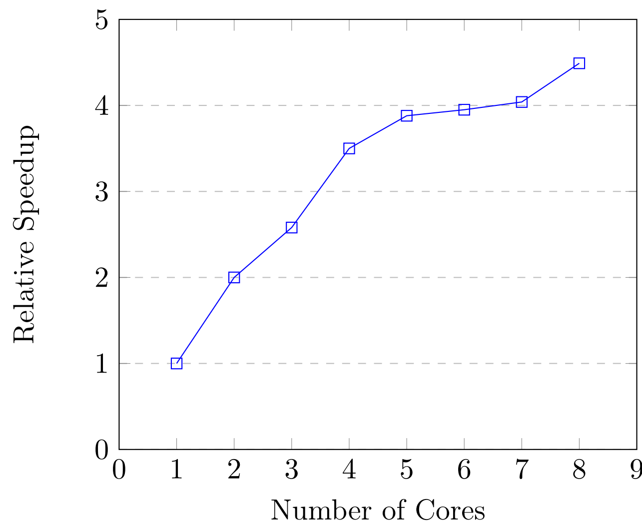

Figure 7.3: Shows the average speedup for different numbers of cores when triangulating 10 million points.

The performance of the different algorithms against each other appears to converge at a large number of points(100 million) where it is expected

that the existing neighbour algorithm will outperform the simpler one which only searches nearby points first. This can be seen in figure 7.4.

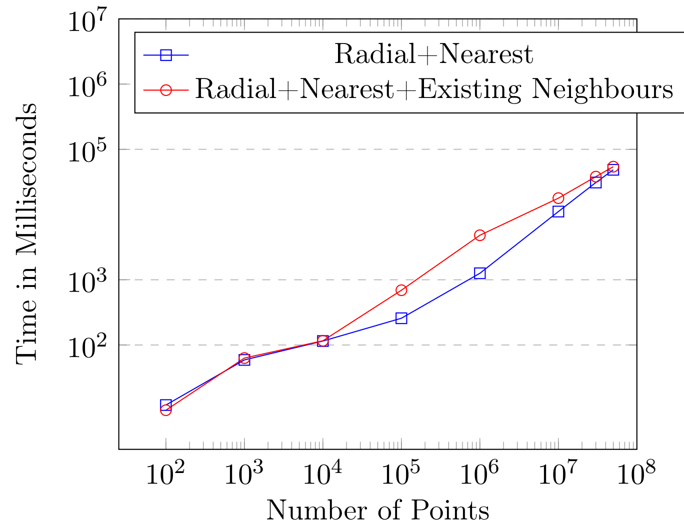

Figure 7.4: Shows the time taken for two of the triangulation algorithms against different numbers of points, the algorithm using the nearest circumcenter radial triangulation and the algorirthm using existing neighbours to find the nearest circumcenter first.

7.3 Program Features

The program in its many iterations has developed to be able to take input from file, generate its own points for triangulation and output these and their triangulation either graphically in a GUI or to standard output. Additionally, there are some diagnostics available for output such as the time used to triangulate. A typical graphical output for triangulation of 100 points looks like the image in figure 7.5.

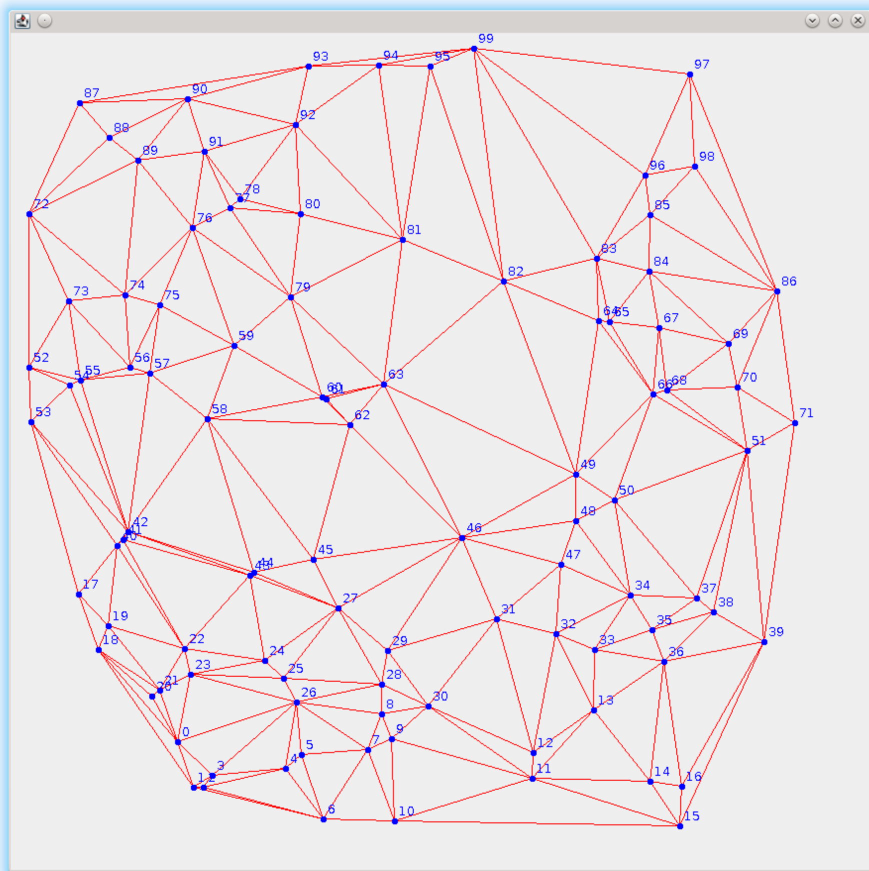

Figure 7.5: Shows typical output from the program for 100 random points(Seed 0 is used here).

Chapter 8

Conclusion

8.1 Goals

The main goals of this project have been the creation of this document and its partner program. As it stands, this document should be sufficient as an introduction to Delaunay triangulation and some of the challenges involved in the design of software to perform the process. While not all the most relevant algorithms have been covered, the rudimentary algorithms for triangulation and the properties and their behaviour have been documented and implemented.

The program itself has a fairly simple structure with no advanced support mechanisms to mix and match algorithms and data structures. Instead, the focus has been on creating consistent interfaces for the data structures and algorithms as a way to allow for future extension of the program's capabilities.

One of the goals was to triangulate 10 million points in a reasonable time, this was met near the end of the project where 10 million points takes between 15-20 seconds on the test machine.

With regards to parallelization, creating the Delaunay triangulation has shown itself to be a task that lends itself easily to parallelization. Many of the algorithms can be implemented to rely on just one mutable point at a time and to disregard any of the mutable data in neighbouring points. This means that race conditions can be completely eliminated and no side-effects will occur in properly implemented concurrent triangulation algorithms. However, since the amount of work searching for neighbours is almost guaranteed to vary from neighbour to neighbour, this disqualifies the problem from being embarrassingly parallel and indicates that implementations on SIMT(Single Instruction Multiple Thread) architectures will suffer performance problems.

8.2 Implementing in Java

Implementing a high performance program in Java posed no serious problem so long as the usage of the standard library and generics was avoided wherever it was possible. Both of these features of the Java language may be very practical, but pose serious performance issues in that the abstractions they create are not easily optimized away. This was obvious in the difference in performance when using primitives vs object wrappers, for example *int* vs *Integer* and the generic containers that only operated on *Integer* objects. When appropriate versions of these were implemented using only primitives, the performance increase was substantial. It is therefore advised to avoid object abstractions and generics when implementing high performance Java code.

Implementing the algorithms and data structures went without much issue except for the recurring problem of validating Delaunay triangles. Due to the usage of the determinant validation method, which caused issues with casting between *long* and *int* types, this caused many bugs and issues that took a great deal of time to track down.

One of the aspects of Java that has the greatest impact on performance is the JIT-compiler that constantly looks for opportunities to optimize the running code. Writing code in a manner that allows the JIT-compiler to optimize the best is not always straightforward and several equivalent variations of expression were tried when implementing solutions to each problem before settling on one that allowed the best optimization opportunities. Some of these techniques include dividing methods up into smaller methods, avoiding any long stretches of code in methods or loops, sometimes leaving duplicate calculations in to allow Java to optimize them away at a later time amongst others.

8.3 Further Work

There are several different things that stand out as possibilities to improve the performance of the program. Especially some of the more efficient algorithms such as the Divide and Conquer algorithms or sweeping algorithms.

Since the definitions used in creating the Delaunay triangulation are highly geometrical, this has meant that spatial sorting and the subsequent creation of subsets of P is one of the greatest optimizations that was done to decrease the runtime of the program. It is probably possible to make even more efficient partitioning strategies that may make especially the querying of circumcircles and half-spaces a more efficient process. Especially being

able to refine the results of subset queries to half-spaces would likely give a decent performance boost.

The algorithms implemented in the program are fairly simple, with the fastest one, the standard search with limited search scope, running at $O(k^2)$. Where k is the average number of points searched while creating subsets of P. k is independent of n.

Implementing "smarter" algorithms was attempted, but the effort was offset by the optimizations the JIT-compiler found for the simpler algorithms vs the smarter ones, the performance ended up better with the simpler algorithms in several cases. Finding a way to implement the smarter algorithms in a way that the JIT-compiler can use would possibly bring great performance gains, indicated by the performance of the simpler algorithms vs the smarter ones when running Java in interpretation mode where the smart ones were faster.

Another large opportunity for performance gains is to find a way to use the Closest Neighbours property to exploit the fact that 70% of the edges in the Delaunay triangulation are already present. An algorithm exploiting this would find some way to detect the "holes" in the triangulations and fill them in, possibly iteratively or by using properties of the Voronoi diagram. Such an algorithm would also need to be able to skip many of the checks of existing edges to avoid a significant performance penality from checking edges that are already guaranteed valid.

Regarding missing features and bugs, the main obstacle that has not been handled is the case of cocircular points. A solution to the case is presented earlier in the report, but the implementation never handles these cases. Find a way to detect and and handle cocircular points in an efficient manner would be desirable in future work.

In closing, the performance of the program matches expectations but there are still plenty of opportunities for improvement, especially with regard to the algorithms.

Bibliography

[1] Boris Delaunay. Sur la sphère vide, 1934.

[2] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis, 1969.

[3] Arne Maus. Delanay triangulation and the convex hull of n points in expected linear time, 1984.

[4] Arne Maus and Jon Moen Drange. All closest neighbors are proper delaunay edges generalized, and its application to parallel algorithms, 2010.