MSc thesis in Geomatics

Simplification of Massive TINs with the Streaming Geometries Paradigm

Maarten de Jong

2021

TU Delft

MSc thesis in Geomatics

Simplication of Massive TINs with the Streaming Geometries Paradigm

Maarten de Jong

June 2021

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of Master of Science in Geomatics

The work in this thesis was carried out in the:

3D geoinformation group
Delft University of Technology

Supervisors:	Dr. Hugo Ledoux
	Dr.ir. Ravi Peters
Co-reader:	ir. Balázs Dukai

Abstract

The volume and density of geospatial data is constantly increasing as newer acquisition techniques are developed to create higher-resolution datasets and more countries are creating country-wide datasets such as point clouds. However, methods to deal with this increase in the amount of data are lagging behind and the developments in computer hardware are no longer moving fast enough to compensate. Especially the ability to load and process datasets into the main memory of a computer is becoming more difficult. Therefore, to work with these massive datasets more modern techniques need to be used. In this thesis on one such method for dealing with these massive datasets is explored; streaming simplification of geometries.

Streaming of geometries relies on processing small portions of a geometric dataset at a time, only keeping as much information in memory as is necessary, instead of the whole dataset. This method ensures that memory bottlenecks are prevented and ever-increasingly large datasets can be processed. Processing done on a LIDAR dataset can range from creating rasterized surface or terrain models, to creating a triangulation. Triangulations are an imporant starting point for other analysis techniques such as watershed calculations, line of sight, or noise propagation. However, large datasets of high density lead to large triangulations which are still too complex for these techniques to work with.

To create more manageable triangulations it is necessary to simplify them, which means finding a subset of the vertices that best approximates the original surface. This can be done randomly, or feature-aware by taking into account the shape of the dataset that is being simplified and ensuring data is kept where it is relevant. The AHN3 dataset is so large that simplification cannot be done using traditional techniques. Therefore, to achieve simplification of the AHN3 dataset, this thesis focuses on the modification of an existing methodology for creating streaming triangulations by incorporating a simplification step.

The methodology implemented in this thesis consists of determining where a simplification algorithm can be introduced within the existing streaming pipeline, as well as determining which existing simplification techniques are applicable. After defining possible locations for a simplification module within the streaming pipeline, various methods are created and tested on different sizes of dataset. These methods are: random simplification, decimation, refinement, and a novel medial axis transform based approach. For the refinement method a number of different implementations are investigated.

After the initial results are analyzed, the decimation and medial axis transform algorithms are rejected due to extremely long processing times or a lack of accurate results, respectively. The refinement implementations are able to produce accurate simplified triangulations in a reasonable amount of time, where the best performing method hardly introduces a slowdown in processing at all. Using the fastest algorithm (FCFS) 4 billion points are processed in less than 11 hours, against 13 hours without simplification. Compared to other methods the accuracy of the methodology is similar for all of the approaches. However, the number of points that can be processed per second is less when compared to existing research. The best performing algorithm is a novel method defined as First-Come-First-Serve (FCFS), as it has the best ratio between vertex error and computation time.

The results of this research show that creation and simplification of a massive Delaunay TIN is possible and produces acceptable results, achieving RMSE's below 0.2m consistently. Despite simplification, the size of the resulting datasets is still often too large to fit within the main memory of a computer and will still require streaming processing to perform further calculations on larger areas. This suggests that with the current ever-growing datasets it is relevant to explore how all different types of calculations can be performed in a streaming pipeline, possibly allowing these to be chained one after another.

All methods created for this thesis are found here: https://github.com/mdjong1/simpliPy

Preface and Acknowledgements

This thesis marks the end of my time at the Delft University of Technology, and with that opens up a whole new world outside of the place I have had the pleasure to call home for the past eight years. In these years, I have worked on numerous projects and assignments, both in my studies, as well as in extracurricular activities, or while working part-time. However, none of these projects has been as challenging to complete as writing a thesis. Therefore I would like to take this page to thank the people that have supported me in this period. Especially in these strange times of the Covid-19 pandemic this was extremely valuable.

First of all, I would like to thank Dr.ir. Hugo Ledoux for meeting with me on a weekly basis. From the beginning of this thesis nine months ago, right up until the delivery, this has been of great value to me. I can't imagine how many times I said things that made no sense when he heard it, but after some prodding and back and forth we managed to both understand what I was actually trying to say. Without his guidance the quality of this thesis would not have been what it is now, and for that I am grateful.

Furthermore, I want to thank my second supervisor, Dr.ir. Ravi Peters, for taking the time to answer all of my questions about how MAT simplification works and providing valuable input at key points during this thesis. The number of relevant and critical thoughts that come from presenting my ideas and intermediate results to Ravi will continue to drive me in the future to be just as analytical. In this process ir. Balázs Dukai was also invaluable, not only in providing more critical thoughts and feedback on this thesis, but also by having the only comparable research that allowed for a direct comparison to my own results.

In the course of the past years I have always been supported by Laura, my friends, and family. This is something that I appreciate now more than ever before; having people around you that help you unwind when you need it most. Without the countless hours that Laura has put into reading my thesis and providing feedback on my figures, I truly believe this thesis would have been less aesthetically pleasing and pleasurable to read. All that remains now is to wish you, as reader, as much enjoyment in reading this thesis as I have had in creating it.

Contents

List of Figures xi

List of Tables xv

List of Algorithms xvii

Acronyms xix

1 Introduction 1
- 1.1 Delaunay Principle . . . 3
- 1.2 Research Questions . . . 5
  - 1.2.1 Scope . . . 5
- 1.3 Outline of this Research . . . 6

2 Related Work 7
- 2.1 Previous Work in Constructing Massive Delaunay Triangulations . . . 7
- 2.2 Spatial Coherence . . . 10
- 2.3 Streaming Principle . . . 11
  - 2.3.1 Streaming of Geometries . . . 11
    - 2.3.1.1 Sprinkling . . . 12
  - 2.3.2 Finalizer . . . 13
  - 2.3.3 Triangulator . . . 15
  - 2.3.4 Applications of the Streaming Geometries Principle . . . 16
- 2.4 Simplification Algorithms . . . 16
  - 2.4.1 Decimation Principle . . . 17
  - 2.4.2 Refinement Principle . . . 19
  - 2.4.3 Medial Axis Transform . . . 20

3 Methodology 23
- 3.1 Integrating Simplification in a Streaming Pipeline . . . 23
  - 3.1.1 Architecture of Simplification within the Streaming Geometries Pipeline . . . 23
  - 3.1.2 Possible Placement of Simplification in the Streaming Pipeline . . . 24
- 3.2 Use of Simplification Methods in the Streaming Geometries Pipeline . . . 24
  - 3.2.1 Randomized Thinning . . . 25
  - 3.2.2 Drop-Heuristic . . . 25
  - 3.2.3 Refinement . . . 28
    - 3.2.3.1 Greedy Refinement . . . 29
    - 3.2.3.2 First-Come-First-Serve Refinement . . . 30
  - 3.2.4 Combining First-Come-First-Serve Refinement with Drop-Heuristic Decimation . . . 33
  - 3.2.5 Medial Axis Transform Simplification . . . 34
- 3.3 Evaluation Criteria: Parameters that Determine the Optimal Simplification Method . . . 35
  - 3.3.1 Accuracy . . . 35
  - 3.3.2 Computation Time . . . 36
  - 3.3.3 Throughput . . . 36
  - 3.3.4 Memory Usage . . . 37
- 3.4 Dealing with Artefacts on Quadtree Borders . . . 37

4 Implementation and Results **39**
4.1 Evaluation Tools . . . 39
4.1.1 Time measurement . . . 39
4.1.2 Accuracy . . . 40
4.1.3 Memory Analysis . . . 42
4.1.4 Streaming Geometries Visualizer . . . 42
4.2 Engineering Decisions . . . 43
4.2.1 Parallelization . . . 44
4.3 Real-World Datasets . . . 44
4.3.1 Frequent tests (Small-Scale) . . . 45
4.3.2 Less-Frequent tests (Large-scale) . . . 46
4.3.3 One-Time tests (Full-scale) . . . 46
4.4 Results . . . 47
4.4.1 Small-Scale . . . 47
4.4.2 Large-Scale . . . 53
4.4.3 Full-Scale . . . 58
4.4.4 Comparison of Results to other Methods . . . 59
4.5 Artefacts . . . 62
4.5.1 Quadtree Border Artefacts . . . 63
4.5.2 Facade Artefacts . . . 63

5 Conclusions and Discussion **67**
5.1 Research questions . . . 67
5.2 Applicability . . . 69
5.3 Discussion . . . 69
5.4 Future work . . . 71

A Reproducibility self-assessment **73**
A.1 Marks for each of the criteria . . . 73
A.2 Self-reflection . . . 73

Bibliography **75**

List of Figures

1.1 3D Noise Simulation from a Triangulated Irregular Network. . . . 1
1.2 How a 3D TIN models a surface. . . . 2
1.3 Effect of directed simplification methods on a small-scale. . . . 3
1.4 Example of a valid Delaunay triangulation. . . . 4
1.5 Example of an invalid Delaunay triangulation. . . . 4
1.6 Example of a constrained Delaunay triangulation. . . . 4

2.1 Comparison of resulting TIN and contour lines between unsimplified and simplified. . . . 7
2.2 Comparison of execution time and memory footprint between ParaStream, Triangle, and Streaming TIN. . . . 9
2.3 Steiner points being added along sub-tile borders to enforce common triangulation points. 10
2.4 Spatial coherence as inherent property of real-world datasets. . . . 10
2.5 Spatial Coherence in the AHN3 dataset. . . . 11
2.6 Configureation of the streaming geometries pipeline. . . . 12
2.7 Visualization of vertex geometries being streamed into a Delaunay Triangulation. . . . 12
2.8 Why sprinkling is necessary to avoid slivers when creating TINs with streaming. . . . 13
2.9 Quadtree cell structure for an AHN3 tile. . . . 14
2.10 Example of the output stream from the finalizer. . . . 15
2.11 How decimation can provide simplified results within a user specified error threshold. . 17
2.12 How z-error is determined for drop-heuristic decimation in a triangulation. . . . 17
2.13 An example of how edge contraction shifts and merges vertices along the connecting edge. 18
2.14 Streaming simplification of a TIN which shows how features are maintained. . . . 18
2.15 How z-error is determined for refinement methods and which vertex is subsequently inserted into the triangulation. . . . 19
2.16 How a minimum angle requirement affects the degree of simplification in a constrained Delaunay triangulation. . . . 20
2.17 How MASB creates the interior surface of a dataset. . . . 20
2.18 Results of MAT point cloud simplification for various maximum error thresholds (ϵ). . . . 21

3.1 Possible placement positions for a simplification module within the *sst* pipeline. . . . 24
3.2 Randomized thinning of the input dataset as simplification method implemented at position B in the *sst* pipeline. . . . 25
3.3 Drop-Heuristic implemented at position C in the *sst* pipeline. . . . 25
3.4 Output example from the Triangulator. . . . 26

3.5 Example of how stars are loaded into decimation. . . . 26
3.6 Drop heuristic may lead to removal of relevant vertices due to cluster removal. . . . 27
3.7 All types of refinement are implemented at position B in the *sst* pipeline, as shown. . . . 28
3.8 Sample input for all refinement algorithms as received from the finalizer. . . . 29
3.9 Comparison of two First-Come-First-Serve refinement techniques. . . . 32
3.10 FCFS + Decimation simplification implemented at position B in the *sst* pipeline. . . . 33
3.11 Comparison of FCFS refinement against FCFS with decimation. . . . 33
3.12 MAT simplification implemented at position B in the *sst* pipeline. . . . 34
3.13 Variation in output point cloud density when simplifying using MAT simplification. . . . 35
3.14 How linear interpolation of a vertex in a TIN works. . . . 36
3.15 Level of visual artefacts using simplification within a quadtree cell . . . 38

4.1 Example of the output from the *time* module in Linux showing real, user, and sys(tem) time. 39
4.2 Vertex error measured vertically and how a TIN subsequently approximates the real surface. 40
4.3 Heatmap of vertex errors created in QGIS . . . 41
4.4 Example showing what plotting the histogram looks like. Vertical lines added to show relevant values: threshold, median, and RMSE. . . . 41
4.5 Example showing the memory usage plot of a simplification module may look like. . . . 42
4.6 The capability of *sstvis* to render streaming geometries in real-time. . . . 43
4.7 Possible placement positions for a simplification module within the *sst* pipeline. . . . 44
4.8 Vertex data being stored into a cell in parallelization. . . . 45
4.9 Overview of all the datasets used. . . . 46
4.10 Delft-Tiny: Error histograms showing the distribution of error for each method. . . . 50
4.11 Delft-Tiny: Error heatmaps for each method. . . . 51
4.12 Delft-Small: Error histograms showing the distribution of error for each method. . . . 52
4.13 Single Tile: Error histograms showing the distribution of error for each method. . . . 54
4.14 Single Tile: Error heatmaps for each method. Zoom in for more detail. . . . 55
4.15 Two Tiles: Error histograms showing the distribution of error for each method. . . . 56
4.16 Four Tiles: Error histograms showing the distribution of error for each method. . . . 57
4.17 Six Tiles: Error histograms showing the distribution of error for each method. . . . 58
4.18 Eight Tiles: Error histograms showing the distribution of error for each method. . . . 59
4.19 Dataset used for the comparison with Dukai [2020]. . . . 61
4.20 Comparison of results from this thesis vs. Dukai [2020]: Error histograms showing the distribution of error for each method. . . . 62
4.21 Level of visual artefacts using simplification within a quadtree cell . . . 63
4.22 No indication of quadtree border artefacts in a small section within the Eight Tiles dataset using Greedy1. . . . 64
4.23 The two types of artefacts that occur between building roofs and the ground surface of the TIN. . . . 64

4.24 Comparison of the Reactor Institute dome-shaped roof between Greedy1 simplification and the 3D BAG dataset. 65

5.1 Possible placement positions for a simplification module within the *sst* pipeline. 68

A.1 Reproducibility criteria to be assessed. 73

List of Tables

4.1 Statistics on the input datasets. . . . 47
4.2 Delft-Tiny: Results of the simplification methods. . . . 48
4.3 Delft-Small: Results of the simplification methods. . . . 48
4.4 Single Tile: Results of the simplification methods. . . . 53
4.5 Two Tiles: Results of the simplification methods. . . . 56
4.6 Four Tiles: Results of the simplification methods. . . . 56
4.7 Six Tiles: Results of the simplification methods. . . . 58
4.8 Eight Tiles: Results of the simplification methods. . . . 58
4.9 Comparison of simplification results from Single Tile to the results recorded by Isenburg et al. [2006d]. . . . 60
4.10 Comparison of simplification results from Single Tile to the results recorded by Hegeman et al. [2014]. . . . 60
4.11 Comparison of Simplification Results to Dukai [2020] based on the data seen in Figure 4.19. 62

List of Algorithms

3.1 Drop-heuristic decimation 27
3.2 Greedy refinement 30
3.3 First-Come-First-Serve Refinement 31
3.4 MAT Simplification 34

Acronyms

AHN Algemeen Hoogtebestand Nederland . . . 1
RIVM Rijksinstituut voor Volksgezondheid en Milieu . . . 1
TIN triangular irregular network . . . 1
DSM digital surface model . . . 1
DTM digital terrain model . . . 1
GIS geographical information system . . . 2
cDT constrained Delaunay triangulation . . . 3
DT Delaunay triangulation . . . 3
AWS Amazon Web Services . . . 8
RMSE root-mean-square error . . . 8
SSDs solid state drives . . . 8
sst streaming startin . . . 12
DEM digital elevation model . . . 16
EPV error-per-vertex . . . 19
LFS local feature size . . . 20
MAT medial axis transform . . . 20
MASB medial axis shrinking ball . . . 20
FCFS First-Come-First-Serve . . . 30
CSV comma separated values . . . 42
SQL structured query language . . . 42
SSD solid state drive . . . 47
DBMS database management system . . . 70

1 Introduction

The amount of geospatial data collected is growing at a velocity and volume much greater than current computer systems can handle. The combination of sensor data, satellite images, and geotagged social media are only a few of many examples leading to the creation of immense pools of data that are too large to process with the most commonly used techniques [Yao and Li, 2018, p. 192]. The density of these existing geospatial datasets is also increasing as new technologies are created to make higher resolution datasets. The consequence of this is that more storage capacity is needed to store all this data, and that data has become more difficult to work with. The same problem is encountered with the LIDAR dataset containing elevation data for the Netherlands (Algemeen Hoogtebestand Nederland (AHN)), which has grown from 2.5 billion points in 1997, to almost 600 billion points in 2019, a 23,900% increase in overall density [AHN, 2020].

In the Netherlands, the choice is made to split this big dataset into smaller tiles of 6.25 by 5km (31.25km^2) so they are easier to work with. Nevertheless, the size of the files is often still too large to fit in the main memory of modern computers, let alone allow easy viewing or processing data in bulk. This is because these datasets are often larger than 14GB and doing calculations with the dataset requires extra memory overhead. This results in an inability to perform calculations on entire AHN3 tiles, let alone have the processing power to process multiple AHN3 tiles at once.

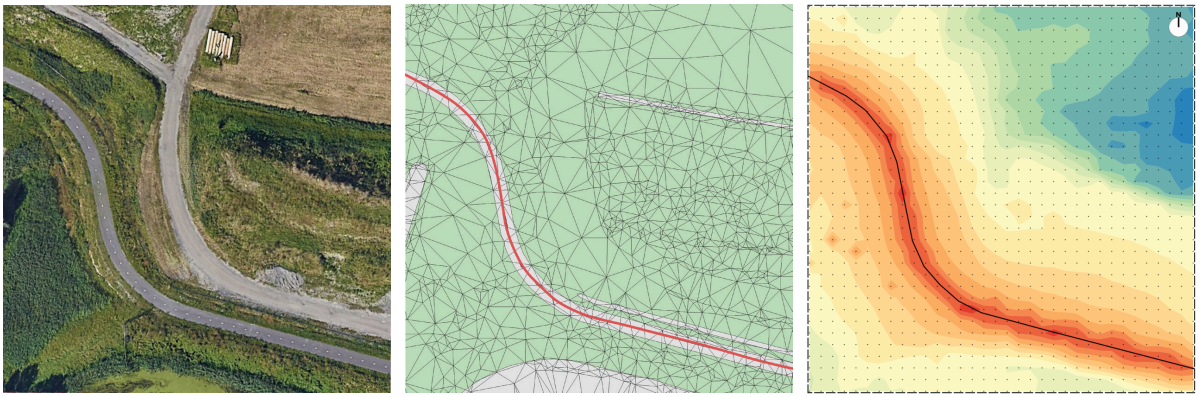


Figure 1.1: 3D Noise Simulation from a Triangulated Irregular Network. Figures from van Rijssel et al. [2020, p. 8, 26].

One of the ways a LIDAR dataset, such as AHN, is commonly processed is to triangulate the points to create a triangular irregular network (TIN). A TIN is a representation of the surface created by triangles between relevant neighboring points. One of the reasons you might want to use such a large triangulation is to model noise propagation in an environment, explored by van Rijssel et al. [2020]. Here, the TIN is used to model how reflections from noise sources traverse an environment and affect the level of noise pollution at another location. An example of how this looks can be seen in Figure 1.1. The Dutch institute for health and the environment (Rijksinstituut voor Volksgezondheid en Milieu (RIVM)) wants to use a similar technique on a single contiguous TIN to model the noise for the entire country. This ensures the surface is always the same for all analyses and is continuous from pollution source to receiver.

One method used by the providers of AHN to reduce the size of the dataset is by providing a rasterized digital terrain model (DTM) and digital surface model (DSM) which have a precision of either 5m or

0.5m. The reduced precision means that the ~15 points per m^2 of the original dataset are reduced to 0.2-2 data values per m^2. This allows the size of the dataset to be drastically reduced and provides datasets that are more usable in day-to-day operations. Examples of operations that can be done on these rasterized datasets include watershed calculations, calculation of line-of-sight, or determining solar irradiance. However, in this process precision is lost due to the cell size being fixed, never allowing for more precision if necessary. It is arguably preferable that datasets with the highest possible accuracy are used when applicable, allowing for down-scaling to be done if necessary. However, considering the current density of the AHN dataset, combined with the lack of a full country-sized TIN, it is very difficult to use this high-precision dataset for day-to-day operations.

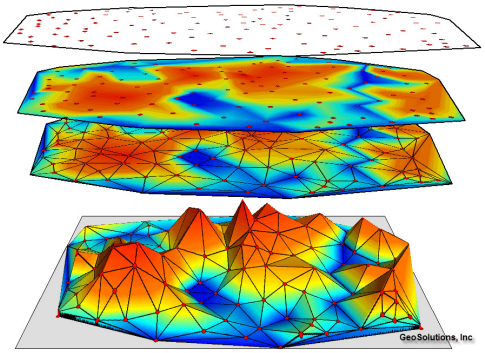


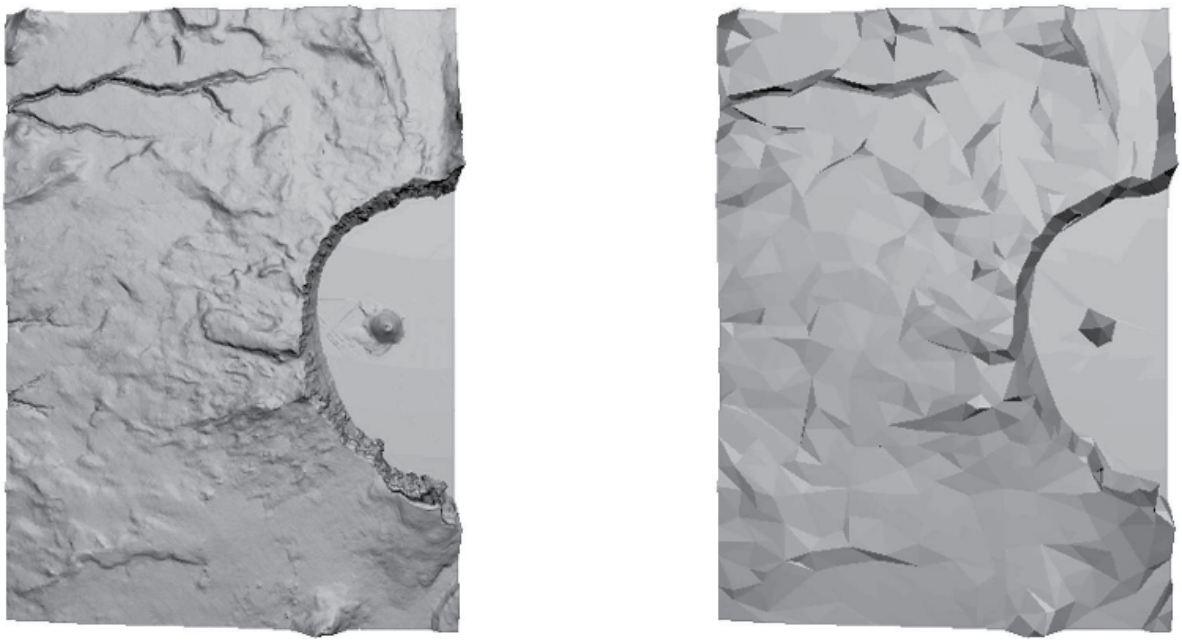
Figure 1.2: How a 3D TIN models a surface by creating representative triangulations between coordinates with a z-value. Figure from URI [2021].

Ensuring precision is not lost can be done by creating a TIN of the available points. Figure 1.2 shows how a TIN is formed, and how points with elevation values can be triangulated to represent a 3D surface. The advantage of creating a TIN is that representative surfaces are created between points, allowing for more complex operations such as interpolation to be performed. However, a disadvantage is that by triangulating an entire AHN tile the file size increases as even more data is added to represent the triangles created.

To overcome the issues of large TINs, it is possible to simplify the triangulations by reducing the number of triangles that represent the same surface, as can be seen in Figure 1.3. This ensures that precision is kept where relevant, while drastically reducing the file size of the TIN. Flat surfaces can generally be represented with fewer triangles without loss of accuracy. While triangulations for 3D models can often be simplified by off-the-shelf geographical information system (GIS) software, these are not capable of handling triangulations that contain many tens of millions of points, as the software will run into memory limitations or crash for other reasons.

One approach taken to creating large Delaunay triangulations is used by Dukai [2020]. This approach relies on splitting one AHN3 tile into nine sub-tiles to create more manageable datasets. Subsequently, triangulations can now be created within each sub-tile because the size of the dataset is reduced. To ensure that the triangulation is still continuous, extra vertices (Steiner points) are added along the edges of all the tiles which are inserted into the triangulation. This ensures a mutual vertex in the triangulation for both sides of a border, thus making the triangulation continuous. Further elaboration on this approach can be found in Section 2.1.

Another way to deal with triangulations containing tens of millions of points without having to load them all into memory is to apply the streaming geometries paradigm [Isenburg et al., 2006b]. The streaming principle reads a large file sequentially; reading a file line by line from start to finish and using a small portion of the entire dataset to determine what needs to be done. This principle can be applied to any type of data processing and is a common technique seen within the field of big data to deal with consistent streams of data. In the case geometries are streamed each line in the file contains a geometric reference; a vertex or an edge. By using this method, the main memory of a computer is never exhausted and various analyses can be done on these large Delaunay triangulations, such as



(a) Terrain model of Crater Lake (199,114 faces).

(b) Simplified model with 999 faces (took 46 seconds).

Figure 1.3: Effect of directed simplification methods on a small-scale. Figure from Garland and Heckbert [1997b, p.7].

simplification, extraction of contour lines, and creation of rasterized data. Theoretically, this allows for infinitely large geometries to be processed as data is continuously loaded into memory, processed, and unloaded from memory to disk. How streaming works in detail is further discussed in Section 2.3.1.

Streaming of geometries only works if the data has high enough spatial coherence, which means that points that are nearby geographically are also close to one another when stored in a file. The principle of spatial coherence is further defined in Section 2.2.

1.1 Delaunay Principle

A TIN is almost exclusively created to adhere to the Delaunay principle, adding a layer of complexity to the calculation used to create these large triangulations. The Delaunay principle states that the circumcircle of each triangle contains no other points. This ensures that triangulations that adhere to this principle maximize the angle of the triangles as well as preventing slivers. Slivers are triangles that are so small and narrow that they hardly have a surface area, making it difficult to perform any operation on them because calculations can result in precision errors. Examples of what a valid and invalid Delaunay triangulation (DT) looks like can be seen in Figure 1.4 and Figure 1.5. Here, vertices A and C in Figure 1.5 are the vertices that violate the Delaunay principle because they are within the circumcircle of vertices ABD and BCD.

Furthermore, it is also possible to create a constrained Delaunay triangulation (cDT), which ensures that specific polygons are maintained within the triangulation. A cDT can be used in cases where building footprints, or other relevant shapes, should be maintained within the output triangulation. This means that the polygons are provided to the triangulation and the triangulator will maintain these polygons as edges of the triangulation. An example of how these polygons are maintained is seen in Figure 1.6, where the line DB is the constraint. This is a similar situation to Figure 1.5, where an invalid DT is presented. This invalid triangulation can be resolved by creating the edge AC marked in red. However,

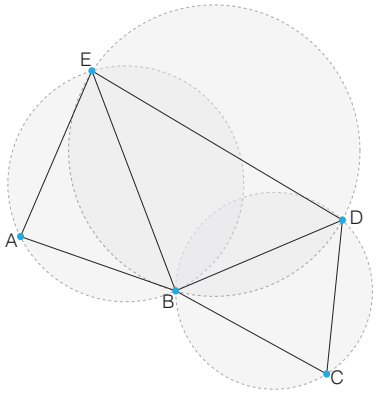


Figure 1.4: Example of a valid Delaunay triangulation. All circumcircles of the triangles contain no other vertices.

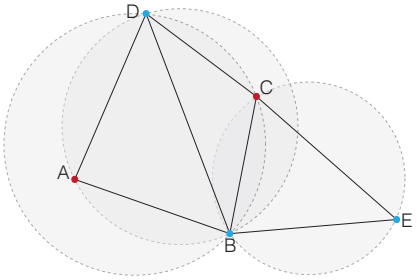


Figure 1.5: Example of an invalid Delaunay triangulation. Two of the circumcircles contain a vertex, marked in red.

due to the added constraint of edge DB this is not possible and the triangulation will not adhere to the Delaunay principle. This is one of the characteristics of a cDT, where the Delaunay principle is often broken to ensure the constraints are met.

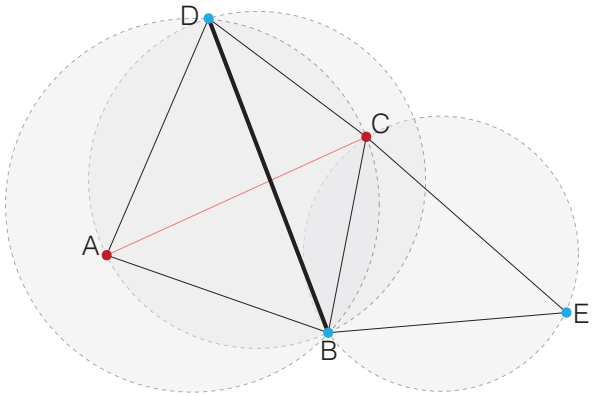


Figure 1.6: Example of a constrained Delaunay triangulation where edge DB is the constraint. To maintain Delaunay edge AC would need to be created instead of DB, but this is not possible.

For the scope of this research constrained Delaunay Triangulations are not taken into account, as the complexity in adhering to the additional polygon constraints is quite high. Furthermore, this research focuses on the streaming geometries paradigm, making the task of constructing a cDT even more challenging as it requires correctly splitting the constraints across quadtree cells.

1.2 Research Questions

The main question for this research is related to how seamless Delaunay triangulations may be created using the entire AHN3 dataset, while also simplifying this triangulation in the process. Combining these two aspects leads to the following main research question:

How can a seamless, simplified, Delaunay TIN for all AHN3 points be constructed using the streaming geometries paradigm?

This research is related to various stages in the process for which the main goal is to achieve a seamless, simplified, Delaunay TIN. One of the supporting questions is therefore:

How can TIN simplification be integrated into the streaming creation of a Delaunay triangulation?

This question relates to the challenges in simplifying the triangulation based on local decisions, as opposed to being able to simplify on a global level. Local decisions are decisions made using only a subset of the entire file, whereas global decisions use all data available in the file. Section 2.4 describes how various simplification methods provide guarantees that are applicable when addressing the global overview of a Delaunay triangulation. As streaming loads and unloads data continuously, using a global metric is not possible and it is therefore necessary to assess how these methods perform when used within a local scope.

For the TIN simplification process, multiple methods are examined and evaluated which are addressed in Section 2.4. Therefore, it is relevant to research which of the simplification methods produces the best results, based on metrics defined in Section 3.3. This leads to another supporting question which reads:

Which TIN simplification method produces the best results when used in a streaming pipeline?

Here, the question focuses on assessing how the simplification methods will yield different results when used in a streaming pipeline. A large part of the performance of the simplification methods depends on user-specified parameters, such as which threshold to use for the error of a vertex. However, optimized tweaking of parameters is out of scope for this research due to requiring a lot of extra computation time to test various combinations.

After creation and assessment of the streaming simplification methods, another relevant aspect is to compare the performance to existing methods. This is based on using the datasets from other researches to quantify how the performance of the methods applied in this thesis relate to the those. The relevant research papers used for comparison are discussed in Section 2.1. In many cases it may not be possible to use the same dataset as described in the existing research due to limited availability, in that case a comparison is made between various metrics to determine how the methods relate to one another. The assessment made using these metrics is discussed in Section 3.3. All this leads to another supporting research question stating:

How does the streaming creation and simplification of a Delaunay TIN perform in comparison to existing methods in terms of execution time, memory usage, and accuracy?

1.2.1 Scope

To limit the size of this research, a number of choices are made that determine the scope within which this research is conducted. One of these choices is that the *sst* program made by Ledoux [2020] is used as the main framework in which the simplification modules are introduced. How *sst* works is further discussed in Section 2.3.1. This streaming geometries pipeline provides a robust starting point for the development of the simplification modules, as the program is open-source, functions well, and is maintained.

Furthermore, this research will be limited to three simplification methods and a control. These methods are further discussed in Section 2.4.

1.3 Outline of this Research

The content of this thesis is separated into four main chapters. Chapter 2 discusses existing research that is relevant to the topic. The main principles are discussed, including how previous large simplified triangulations have been created, as well as information on the chosen simplification methods for this research.

Chapter 3 is used to elaborate on the design choices that are made based on existing research. This chapter goes into depth on specifics of why choices are made, as well as discussing initial findings.

Chapter 4 deals with *how* these choices can be used. This chapter elaborates on the technical implementation of this research. Furthermore, this chapter also shows all the results obtained.

Finally, Chapter 5 provides a conclusion and discussion by answering the research questions and providing depth on the usability of this research. This chapter also reflects on the results and suggests future work that is possible with this research.

2 Related Work

To better understand the possibilities and limitations of streaming geometries and streaming simplification, this chapter discusses the previous findings from other related research. Firstly, the streaming of geometries is tackled and other relevant methods for dealing with large Delaunay triangulations are reviewed. After this, the TIN simplification methods that are applicable in the context of the streaming geometries paradigm are presented.

2.1 Previous Work in Constructing Massive Delaunay Triangulations

Besides the works of Isenburg et al. [2006d] and Constantin et al. [2010] related to simplifying Delaunay triangulations by streaming geometries which are explained in Section 2.3, there are very few methods that do not rely on the streaming geometries paradigm to create massive Delaunay triangulations. Furthermore, there is a lack of relevant research that aims to simplify a large Delaunay triangulation while it is being created. Many researchers focus merely on the creation of large Delaunay triangulations without including a simplification step.

Isenburg et al. [2006d] focus on the streaming creation of contour lines, but choose to implement a simplification step to make it easier to create clean contour lines. The inner workings of this simplification step are omitted though a stated simplification of 90% is achieved with no error metric provided. Figure 2.1 shows the result of the contour line extraction before and after simplification. Despite the result in Figure 2.1 hardly changing between with and without simplification, it is difficult to state whether the simplification done by Isenburg et al. [2006d] is randomized simplification or a directed simplification method.

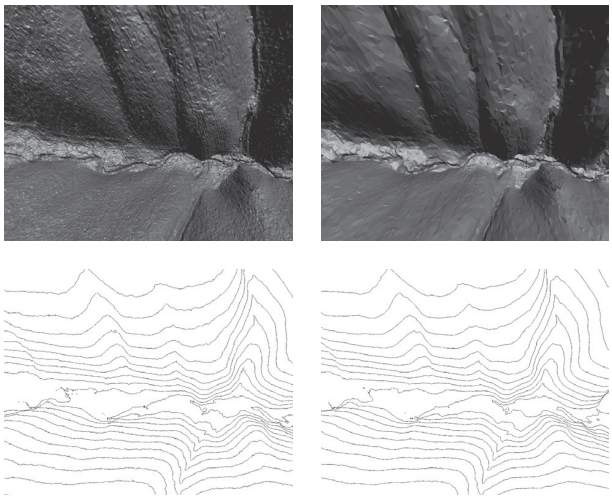


Figure 2.1: Comparison of resulting TIN and contour lines between unsimplified (left) and simplified (right). Figure from [Isenburg et al., 2006d, p. 4].

The main reason it is not possible to determine what kind of simplification method is used in Isenburg et al. [2006d] is because randomized thinning creates a near-constant error across the TIN, which may result in contour lines that are nearly the same as before. Nevertheless, Isenburg et al. [2006d] also state

that the simplification module takes the most CPU time of the entire pipeline, which suggests many calculations are done within this module. Therefore it can be assumed a directed simplification method is used to perform simplification in this streaming pipeline to create contour lines. This means that for later reference the method used by Isenburg et al. [2006d] for simplification prior to contour line extraction can be best compared to the directed simplification methods created for this research, further discussed in Section 2.4. Unfortunately no binary or code is available of the exact method, thus the statements made in Isenburg et al. [2006d] for the number of points processed in a period of time are used to perform comparisons in Table 4.9.

Hegeman et al. [2014] is one of the few methods for creating massive Delaunay TINs and explores how large Delaunay triangulations can be created using computing clusters available from Amazon Web Services (AWS). The approach of Hegeman et al. [2014] uses a distributed computing cluster with memory capacities between 128GB and 2.2TB to process datasets between 12GB and 95GB. The methodology is focused towards the practicalities of computing a coherent Delaunay triangulation in a distributed cluster by splitting and merging data efficiently. However, the research does state that it achieves a 0.069m root-mean-square error (RMSE) with 32% of the vertices removed, and a 1.9m RMSE with 81% of vertices removed. Considering the high RMSE of 1.9m with 81% of the vertices removed it is likely that the simplification method used by Hegeman et al. [2014] is based on a simple thinning method which probably relies on randomized thinning or removal of $1/n$ points. Therefore, the performance of this distributed cluster computing method is best compared to randomized thinning (Section 3.2.1) in a streaming pipeline.

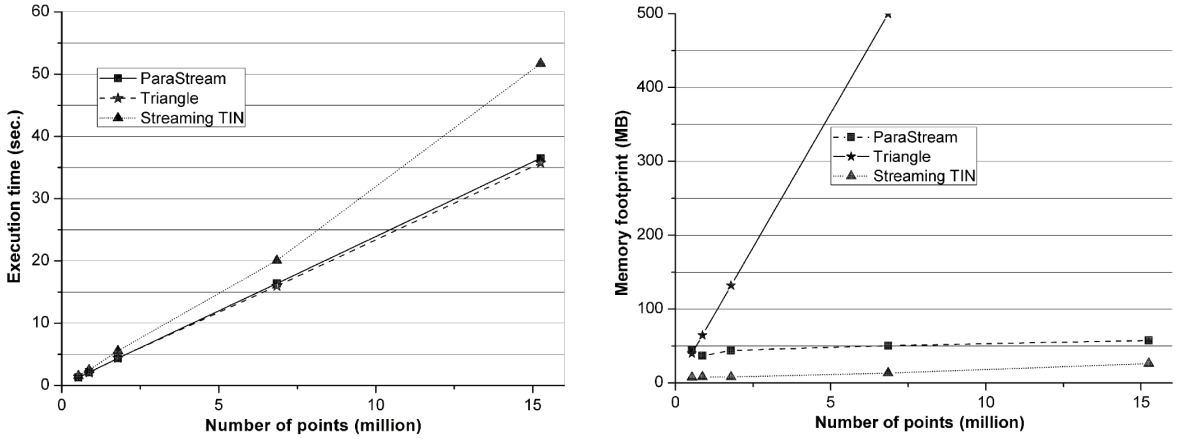
Another approach is the out-of-core method which has been applied by Agarwal et al. [2005] to create constrained Delaunay triangulations. cDTs use a polygon to indicate lines that must be present in the output triangulation, as explained in Section 1.1. An out-of-core algorithm aims to organize the data on disk in such a way that it can be optimally read into the internal memory. The goal is to ensure that the next data block to be read is the next physical block on the disk which ensures minimal seeking of the storage device. This is something that has been relevant for mechanical drives but is now being quickly outdated by the widespread availability of solid state drives (SSDs) which feature a linear access time. Therefore Agarwal et al. [2005] will not be taken into account as a relevant, modern, option for creating massive Delaunay TINs. Nevertheless, the approach used to create a cDT provides insight into possible future work in the creation of constrained triangulations within the streaming pipeline. This future work is discussed in Section 5.4.

Agarwal et al. [2005] state they are able to create a cDT from 10GB of real-life LIDAR data using 128MB of memory in roughly 7.5 hours [Agarwal et al., 2005, p. 357]. This is done by taking a heavily mathematical approach towards creating an I/O efficient algorithm which initially creates a triangulation of the dataset using a user-specified block size, which is followed by introducing the polygon constraints into this triangulation. The block size specifies how much data should be stored in each segment on the disk. By determining the relation between the completed blocks, these blocks can be stored in the correct geospatial order allowing for fast access when adding the polygon constraints later on. For this, Agarwal et al. [2005] use a Hilbert order to create the geospatially efficient storage. At the end of the processing the result is a set of blocks on disk which are in the correct geospatial order to be written to a readable file for output. This writing to file is much faster on a mechanical drive compared to other methods also using mechanical drives because the drive requires minimal movement to write the data blocks in the correct order.

Another proposed method relies on sorting the dataset within the file itself prior to processing it. Buchin and Mulzer [2011] discuss how they create a Delaunay triangulation by initially sorting all the vertices within the dataset on disk and subsequently triangulating the sorted dataset. Triangulating sorted points ensures that subsequent points read from the file are always nearby to one another, reducing the number of very large triangles created that require insertions within these triangles at a later moment. The approach is interesting, though sorting nearly 600 billion points for the entire AHN3 is a task that will take too long to complete and is therefore not a feasible method for my research. Furthermore, as is discussed in Section 2.2, real-world datasets have inherent spatial coherence which prevents the need to sort them prior to processing.

A different approach is proposed by Wu et al. [2011] in which the multi-core capability of modern CPU's is utilized to divide and conquer the workload across different cores. As can be seen in Figure 2.2, the memory usage is increased but execution time is decreased. Wu et al. [2011] achieve this by spreading their workload across multiple CPU cores, effectively increasing the number of computations that can be done. Their approach can be combined with Funke et al. [2019] who discuss how to best divide the workload across the cores. Funke et al. [2019] create a methodology that ensures data is logically separated and workload sizes are equally spread. Their methodology is relevant because it ensures that each core is provided with a task that can be solved and can be performed in a similar amount of time to other cores.

The approach of Funke et al. [2019] builds upon the multi-core workflow discussed in Funke and Sanders [2017], and thereby provides an improvement upon the original research by Wu et al. [2011]. Funke et al. [2019] is the first research to combine both streaming TIN with multi-core processing as described by Wu et al. [2011]. This approach is very relevant as it ensures that the tasks within the streaming pipeline that can be parallelized run much faster. Examples of this are the parallelization of the three passes within the finalizer, as discussed in Section 5.4, or the simplification of multiple quadtree cells in parallel as discussed in Section 4.2.1. As such, similar principles to the ones described by these three research papers are applied to improve the execution time of the simplification methods created for this research. The methods are discussed in Section 3.2 and how multi-core processing is combined with these methods can be found in Section 4.2.1.



(a) Execution time

(b) Memory footprint

Figure 2.2: Comparison of execution time and memory footprint between ParaStream, Triangle, and Streaming TIN. Figures from Wu et al. [2011, p. 1361, 1362].

Lastly, Dukai [2020] has recently taken an approach towards processing the entire AHN3 dataset into a TIN by using additional points along tile borders, called Steiner points, shown in Figure 2.3. This method splits a single AHN3 tile into nine sub-tiles for easier processing and storage. The method consists of interpolating the Steiner points along the sub-tile borders every ten meters. By doing this, tiles that share a border have mutual points that can be used to connect the triangulations of those tiles together. This method is successful in producing a country-sized TIN, while keeping each tile available separately. Therefore, the results of this method can be used as a comparison for vertex count and accuracy when comparing to the result of this research.

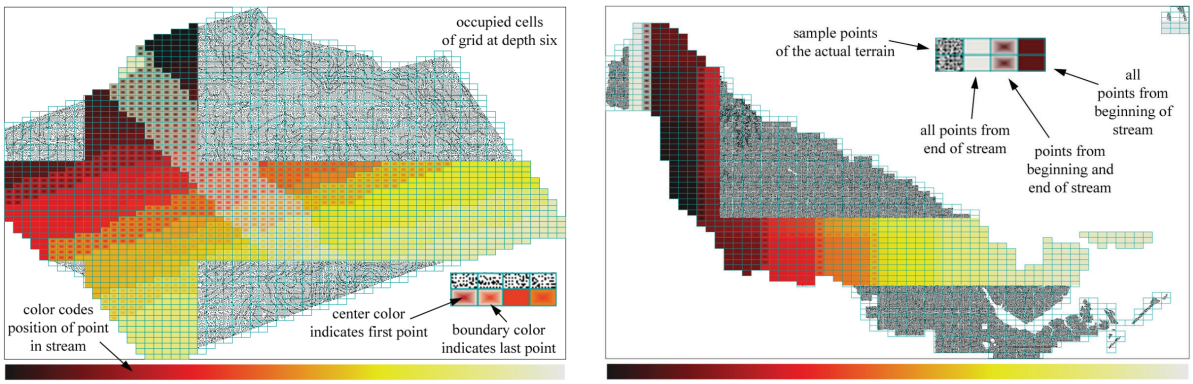
Dukai [2020] uses multiple processes to achieve the simplified TINs. To start, the AHN3 tiles are split and the borders are densified; adding Steiner points every ten meters. After this, each sub-tile can be processed by 3dfier which is configured to simplify the dataset to a maximum error of 0.3m using greedy refinement [Ledoux et al., 2021b]. Finally, duplicate points along the edges are removed and the result is stored to a PostGIS database. This methodology relies on a number of different programs that are chained together, similar to the streaming geometries paradigm. However, in this case the cell size is determined by the size of the sub-tiles and the streaming is done by chaining various different programs

Figure 2.3: Steiner points being added along sub-tile borders to enforce common triangulation points. Figure from Dukai [2020].

together. It should be noted this chaining is done manually as the entire AHN3 dataset is processed for each stage before moving on to the next step. Therefore it is not possible to state what the processing time of this methodology is and a comparison can only be made for the outcome of this approach, which is performed in Table 4.11.

2.2 Spatial Coherence

Spatial coherence is a property which correlates the proximity of geometries in their 3D space with the position of their representation when being streamed [Isenburg et al., 2006b, p. 1]. This property ensures that when creating a triangulation, nearby points can be triangulated and finalized on a frequent basis. By having high spatial coherence it ensures that few vertices need to be kept in memory and the application can continue to function memory-efficiently. [Isenburg et al., 2006b] has shown how spatial coherence is an inherent property of real-world datasets by analyzing the spatial coherence for two datasets seen in Figure 2.4.

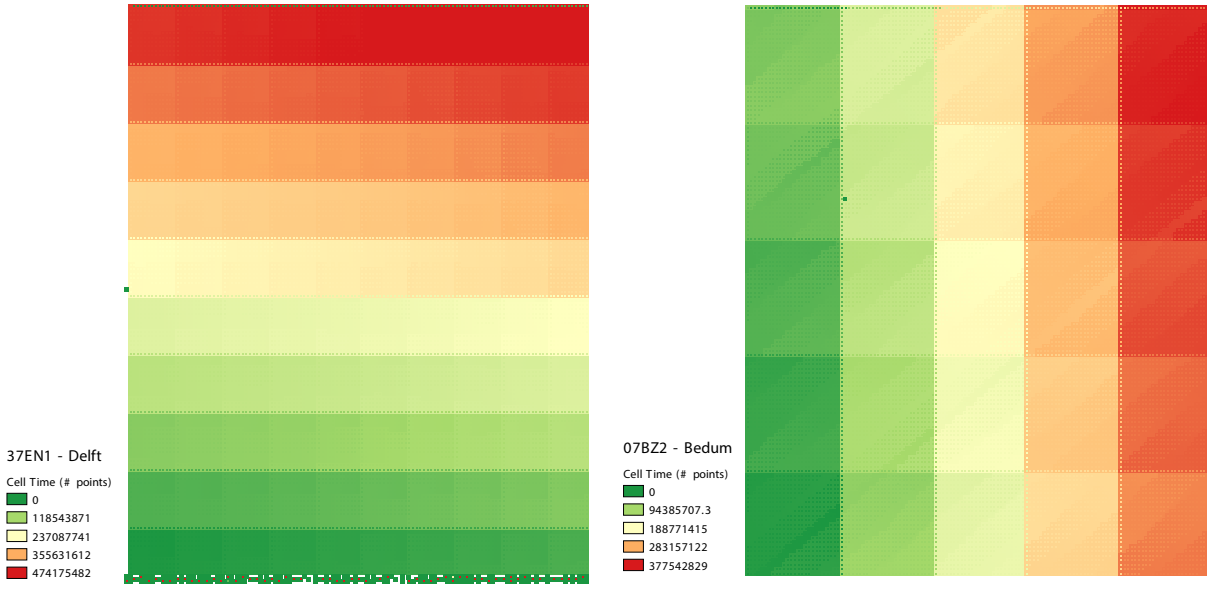


(a) 6 million point Baisman Run dataset in Broadmoor, Maryland

(b) 500 million point Neuse river basin dataset in North Carolina

Figure 2.4: Spatial coherence as inherent property of real-world datasets. Adapted from Isenburg et al. [2006b, p. 3].

To determine spatial coherence, a dataset can be split into cells that are used to show how correlated the points within that cell are. After this, the dataset is streamed and it is determined within which of these cells the point lies. The 'time' is measured by how many vertices have been processed at that moment. When all points in the dataset have been processed, the result is a raster grid with cells that contain a start time and a last update time. This provides all necessary data to determine the spatial coherence of that dataset. Using this approach and applying it to AHN3 shows that for various AHN3 tiles located in different locations in the Netherlands this property is valid. Two examples of this can be seen in Figure 2.5.



(a) 37EN1 (Delft, Zuid-Holland)

(b) 07BZ2 (Bedum, Groningen)

Figure 2.5: Inner cell color indicates time in stream of first point in that cell, cell border indicates time in stream of last point in cell. Both tiles show a few outliers, but overall spatial coherence is very high for the tested AHN3 tiles.

2.3 Streaming Principle

Streaming consists of two (or more) programs that are connected with a *pipe* operator. This is an operator that takes the output of the previous program and uses that as input for the next program, an example of this is **echo "Hello World" | wc –w**. The pipe operator is denoted as |. This will output "Hello World" to the pipe operator, which will then be fed into *wc* (word count) and tell us how many words are in the input sentence (two). This same principle is used to chain applications together that in- and output geometric data.

The way these pipes function is that data is sent continuously as the programs in the stream are running. Therefore it is not necessary to wait for one of these programs to finish running in its entirety before being able to continue. In the simple case stated above, where the number of words is counted, the same operation can be applied to a continuously growing file and the word count will continue to count the words in that file until it is finalized.

2.3.1 Streaming of Geometries

Streaming of geometries is used to process geospatial datasets that are too large to fit in the main memory of a computer. Initially presented by Isenburg and Gumhold [2003], it allows for the processing of

large datasets on commodity hardware. Though powerful hardware is more common nowadays, the size and density of datasets continues to grow faster than the capability to process these datasets. The two programs used for streaming are a finalizer and a triangulator, seen in Figure 2.6, both of which are explained more elaborately in the following sections.

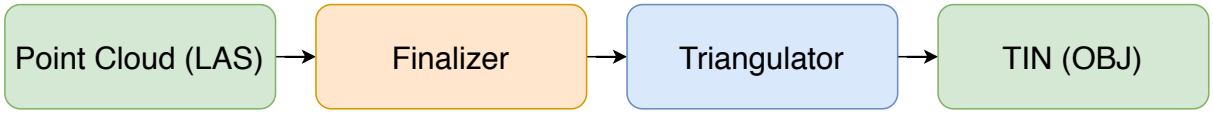


Figure 2.6: Configuration of the streaming geometries pipeline to create large Delaunay triangulations as proposed by Isenburg et al. [2006b], and later used by Ledoux [2020].

The streaming geometries principle presented by Isenburg and Gumhold [2003] is later elaborated on in Isenburg and Lindstrom [2006], and Isenburg et al. [2006b]. These three researches together create a methodology in which large meshes can be rendered and processed while streaming. In the latter it is also discussed how Delaunay triangulations can be created while streaming. Isenburg and Gumhold [2003] aim to create a methodology to visualize existing large-scale meshes, whereas the other two researches aim to apply an operator on the streaming mesh and target Delaunay triangulations more specifically. The third manages to create a Delaunay triangulation from 11GB (500 million points) of LIDAR data in less than an hour using 70MB of memory, the process and result of this can be seen in Figure 2.7. This methodology consists of two main elements: a finalizer and a triangulator. Both of these elements are discussed in more detail in the following sections.

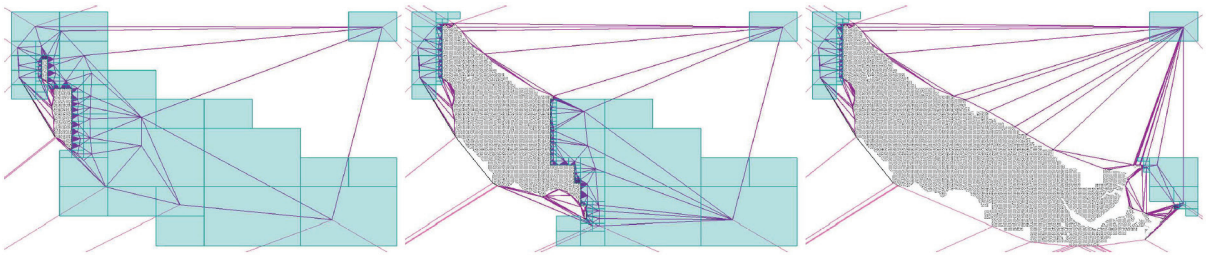


Figure 2.7: Visualization of vertex geometries being streamed into a Delaunay Triangulation. Blue quadrants show unfinalized space where points will arrive in the future. Black points have already been written to disk or piped to the next application. Adapted from Isenburg et al. [2006b, p. 1].

2.3.1.1 Sprinkling

Sprinkling is the process in which some points are promoted to the beginning of the stream, which means that prior to triangulating these points are already present. The sprinkling process is performed as part of the finalizer (discussed in the following section). The reason that sprinkling is necessary is that in some (rare) cases points are read from the input file in such a way that triangulating requires moving of all previously created triangles due to creating slivers. These sprinkle points are inserted to ensure that there is always a larger triangulation present, thus preventing these slivers from being created. A possible scenario in which sprinkle points are necessary to prevent unnecessary computations can be seen in Figure 2.8. This figure clearly shows a large number of slivers that will result in many operations to correctly adjust these triangles upon inserting a new vertex, followed by a triangulation including sprinkle points showing how these slivers are avoided.

Sprinkling can be done by promoting a certain percentage of points to the top of the stream as they are released from the finalizer, discussed in Section 2.3.2. Another method is to promote one point per cell, ensuring that there is always at least one randomized point present in a cell. streaming startin

(sst) functions by promoting a certain percentage of points (0.1%) to the top of the finalization stream. Isenburg et al. [2006b] relies on efficiently promoting a single point per cell, though it is stated that these points are only released as soon as the quadtree cell becomes relevant [Isenburg et al., 2006b, p. 6]. Releasing when a quadtree cell becomes relevant is more complex than promoting all sprinkle points to the top of the stream, but it prevents releasing a large number of points at the start of the streaming pipeline which improves the overall processing time.

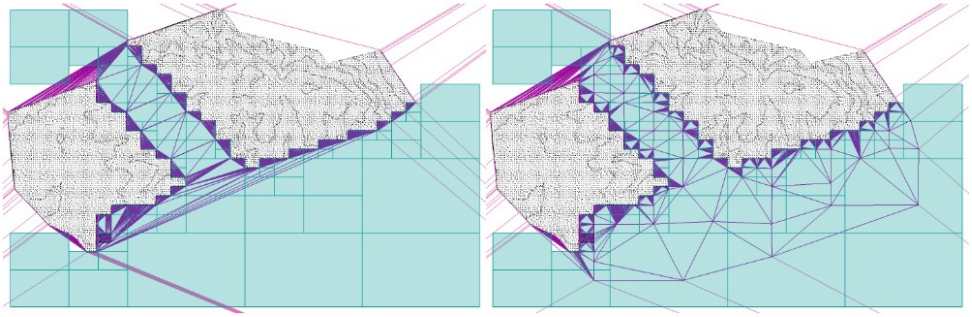


Figure 2.8: Skinny temporary triangles (left) are avoided by lazily sprinkling one point into each unfinalized quadrant at each level of the evolving quadtree (right). [Isenburg et al., 2006b, p. 6]

2.3.2 Finalizer

With its misleading name, the *finalizer* is the first application in the streaming geometries pipeline that a file is piped through, as can be seen in Figure 2.6. The task of this operator is to do all necessary initialization steps before processing begins. The reason it is called finalizer is because it releases cells in chunks of a specified dimension and thus 'finalizes' these chunks. How finalization and releasing of chunks works is further explained in the rest of this section. The finalizer relies on passing over all the data in the input file(s) to obtain the necessary information to allow processing of the data to start. This is done in three passes and can therefore take a while to initialize for larger datasets, as each pass reads every point in the file(s). Each pass has its own function and uses information from the previous pass. Therefore, it is not possible to everything in a single pass (yet), though advancements could be made to reduce the number of passes and their respective complexity.

What the finalizer initially does is retrieve the number of points that are in the file, which is used to determine how many points need to be sprinkled. If this information cannot be retrieved from the header of the file, this is the first full pass of the input file. A specified percentage of points, around 0.1%, is chosen at random from the file to be released to the rest of the pipeline immediately. Furthermore, the bounding box is determined by determining what the minimum and maximum x, y, and z value is. These values are needed to create the quadtree structure that is used to subdivide all points. In Figure 2.9 an example is shown of what this quadtree structure may look like for an AHN3 tile.

After the sprinkling is completed, the finalizer moves on to a second pass of the data in which the number of points per quadtree cell is summed. Furthermore, points can be filtered at this stage dependent on their classification. Filtering is done in the second pass, as opposed to the first pass, because with the current structure of sst this implementation meant less adjustments. Preferably, however, the filtering is done in the first pass, which also counts the number of points, to ensure the correct number of points is taken into account. Filtering means that, for example, all non-ground points can be rejected to create TIN of the surface. Other filtering combinations are also possible depending on the classifications present in the dataset. The calculation for determining in which x and y cell a point lies is seen in Equation 2.1. This second pass creates a grid of dimensions cell-size x cell-size. Each cell now contains the value of the number of points that are present in that cell.

$$cell_x = floor((x_{pos} - min_x)/cellsize) \qquad cell_y = floor((y_{pos} - min_y)/cellsize) \tag{2.1}$$

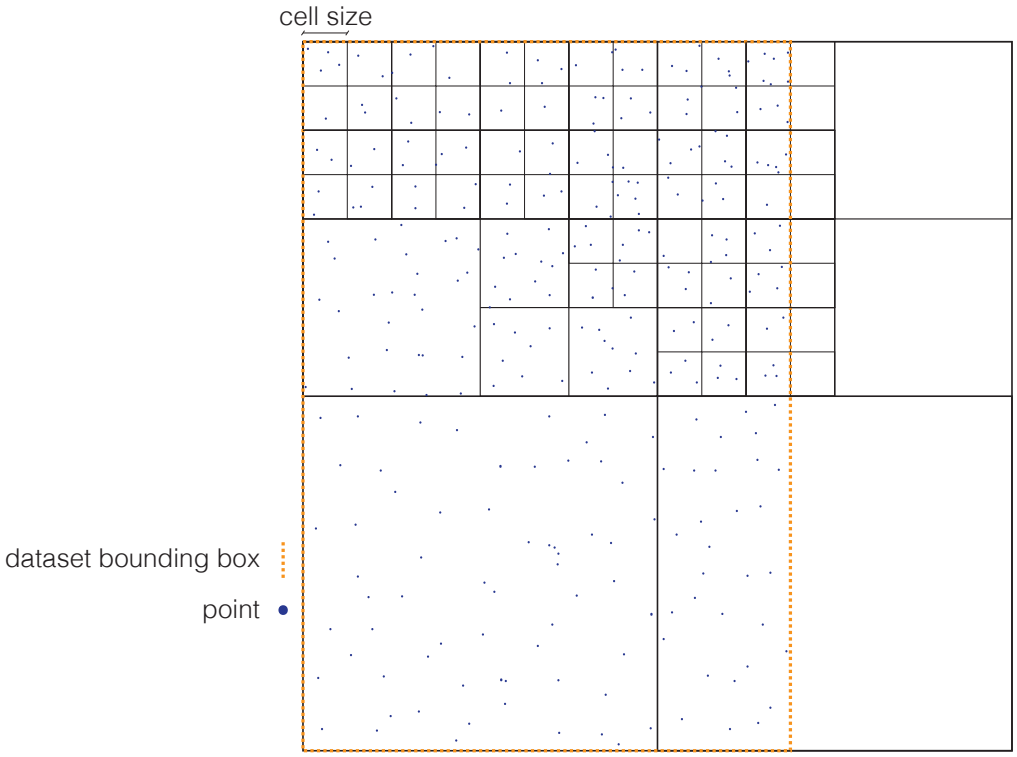


Figure 2.9: Cells outside the bounding box of the dataset are empty and thus finalized at the start. When four neighbors are finalized they are merged into a larger cell.

The regular grid is initially used to determine the smallest cell size within which vertices can be subdivided. Using the regular grid it is possible to determine when a cell is finalized and can be merged with its parent cell. By using this regular grid it is always possible to use a simple calculation, seen in Equation 2.1, to determine within which cell the point lies. As soon as cells start being merged into larger cells, as seen in Figure 2.9, it requires much more tracking of parent and child cells to determine within which cell a point lies. Therefore, the choice is made within sst to use this regular grid which is only merged with a parent upon finalization of all four child cells. This ensures that no points will enter those quadtree cells at a later moment in time, reducing the amount of bookkeeping that is necessary to keep up with cell dimensions and positions of points.

An example output of the finalizer is seen in Figure 2.10. Figure 2.10 is also used to reference the line numbers indicated in the text to provide a textual and visual idea of the content of this module. In the third and final pass of points through the finalizer, each point is released per quadtree cell to the pipeline for further processing. This is done by initially writing the background information to the pipeline, allowing the next processes to use this information. This background information relates to: the number of points in the entire dataset (line 3), the number of cells (line 5), the size of each cell (line 7), and the bounding box of the dataset (line 9). All the background information is relevant for either the simplification method that follows, or the triangulator to create an initial grid. After this information has been written to the next module of the pipeline, all quadtree cells that have no points can be released as they require no further processing and can be finalized immediately. Subsequently, the sprinkle points are released since these are necessary to ensure slivers don't occur. Lastly, the points in each quadtree cell are released cell-by-cell, with one vertex on each line of output data (lines 11-14 and 18-19). After all the points in a cell are released, the cell can be finalized with a finalization tag (line 17). When all the quadtree cells have been released to the stream the finalizer is finished with its task.

Furthermore, it is important for the finalizer to operate with a minimal memory footprint as the entire streaming pipeline is meant to easily fit in the main memory of a computer. The memory footprint of

```
1   # sstfin
2   # number of vertices in dataset
3   n 212550
4   # number of cells in x and y direction
5   c 2
6   # width/height of each cell
7   s 50
8   # bounding box of dataset
9   b 84600.000 447000.000 84699.998 447099.999
10  # vertices
11  v 84693.724 447051.214 1.410
12  v 84628.038 447085.693 9.739
13  v 84684.830 447073.781 0.877
14  v 84650.951 447082.913 0.584
15  ...
16  # finalizing cell x=1 y=0
17  x 1 0
18  v 84699.781 447050.264 1.284
19  v 84699.508 447050.050 1.307
20  ...
```

Figure 2.10: Example of the output stream from the finalizer.

the finalizer has been measured to be a maximum of 40MB, which is well within bounds. Data held in memory mainly relates to the input parameters and characteristics of the input data, i.e. which files to use as input, the id's of vertices to sprinkle, which point classifications to keep, and the bounding box of the dataset.

2.3.3 Triangulator

While the finalizer is still busy releasing information to the pipeline, the triangulator can start processing all data that it receives immediately. The advantage of using a streaming pipeline lies here, as the triangulator is constantly processing each vertex that the finalizer releases and finalizing the cell when it receives a finalization tag.

The triangulator functions differently from the finalizer, as it reads all its data from the previous module and thus has no knowledge of the entire dataset besides what it is provided by the finalizer. The input of the triangulator is the output of the finalizer, as shown in Figure 2.10. Using the provided tags containing the background information (lines 1-9), the triangulator is able to create the same cell grid as is made in the finalizer. For each point that the triangulator then receives it inserts it into the correct cell.

When it receives a finalization tag in the shape of "$x\ cell_x\ cell_y$" (line 17), the triangulator knows it is allowed to finalize all the points in that cell, as no new ones will enter it later on. Finalization can then be done when there are no edge cases such as vertex edges being on the convex hull, or if the circumcircle of a triangle is near the bounding box of the dataset. If these checks pass, the triangulation that has been created of all points in the cell is written to the next module in the pipeline. The triangulator is generally the last module in the pipeline, which means that all the results from the triangulator are written to disk and thus released from memory, keeping the memory usage minimal.

The triangulator writes the resulting triangulation in the OBJ file format which is easily readable by a number of programs. This format outputs vertices and their corresponding edge relationships into a string format on file. Edge relationships are defined as 'faces' and represent each triangle that is present

in the triangulation. Faces in OBJ may also be defined as quadrilaterals as opposed to triangles, though only triangles are used to store the results of the triangulations.

As with the finalizer, the triangulator should also use a limited amount of memory to stay within bounds of the main memory of a computer. The amount of memory used by the triangulator has been measured to be around 50MB maximum, which is used to store all active vertices and their triangulations.

2.3.4 Applications of the Streaming Geometries Principle

Isenburg et al. [2006b] also looked further than the creation of triangulations, by exploring possibilities in streaming geometries by looking at compression [Isenburg et al., 2006a], creation of contour lines [Isenburg et al., 2006d], and creation of digital elevation model (DEM)s [Isenburg et al., 2006c]. With these resources a majority of the desired techniques for processing large LIDAR datasets by streaming is created. One issue is that no corresponding code or binaries are released for most of these researches. Ledoux [2020] has since recreated the main streaming pipeline proposed in Isenburg et al. [2006b] and made the project available open source. This includes the finalizer, triangulator, and a statistical analysis tool. However, further tools similar to those described are not readily available.

2.4 Simplification Algorithms

Considering that simplification is the main goal of this research, a number of simplification methods are used to assess which method will best answer the research questions. Most of the methods discussed are not designed to be applied with streaming geometries, thus adaptations are necessary to be able to use them within the sst streaming geometries pipeline. The versions of these methods as used for this thesis are discussed in Section 3.2. For this thesis three base simplification types are considered, which are:

1. Decimation
2. Refinement
3. Medial Axis Transform

Simplification algorithms can be applied *globally*, as regular GIS software does (described in Chapter 1), within which the algorithms use the entire dataset to create an optimal solution using all the available data. By using a global algorithm a maximum ϵ vertex error can be guaranteed as all points in the dataset are usable by the simplification algorithms. This provides security that the desired ϵ is always achieved.

As opposed to regular GIS software packages, when using a streaming geometries pipeline it is only possible to access the subset of data held in memory at that moment in time. Therefore, a streaming simplification algorithm will never be able to use a global metric to determine what the maximum ϵ vertex error is for the resulting triangulation. Instead, the simplification algorithms used while streaming rely on a *local* metric, namely the ϵ within the subset of data that the algorithm is processing at that moment. For some of the following simplification algorithms the data used by simplification is the content of a quadtree cell, for other simplification algorithms this is a user-specified number of points that should be used. Because of this locality in simplification a global ϵ cannot be guaranteed, thus increasing the probability that the user-specified maximum vertex error is not be achieved for the entire dataset.

2.4.1 Decimation Principle

Decimation is the first main type of simplification that can be done to simplify TINs. This type of simplification works by taking a full triangulation and removing all vertices that have an error value that is beneath a user-specified threshold. An example of how this error threshold works can be seen in Figure 2.11. Here, it is shown that the curved surface is approximated within ϵ threshold to retain its original features.

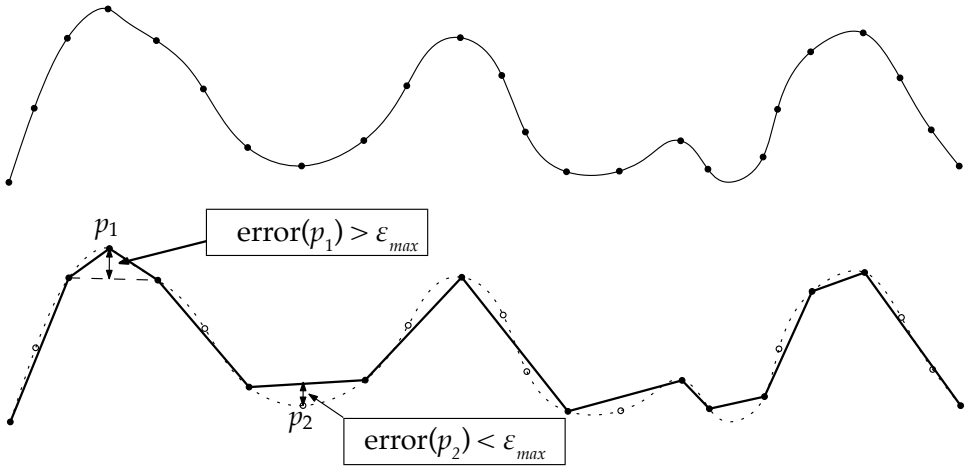


Figure 2.11: How decimation can provide simplified results within a user specified error threshold. ϵ_{max} denotes the user-specified vertex error threshold. Figure from [Ledoux et al., 2021a, p. 84]

The main method within this category is the drop-heuristic method developed by Lee [1989]. This method works by determining the error of each vertex in a triangulation if it were to be removed. Subsequently, the vertex with the least error of all the vertices is permanently removed from the triangulation. Error is measured by removing the vertex from the triangulation; linearly interpolating what the value of the vertex is with it removed; then re-adding the vertex to the triangulation. A visual representation of how this process works can be seen in Figure 2.12. The error value is determined to be the difference between the actual z-value of the vertex and the interpolated value. After all vertices are measured, the vertex with the least z-error is permanently removed from the triangulation as long as that z-error is above a certain threshold. If the smallest z-error available in the entire triangulation is above the threshold, the triangulation has been simplified sufficiently and the decimation process can finish.

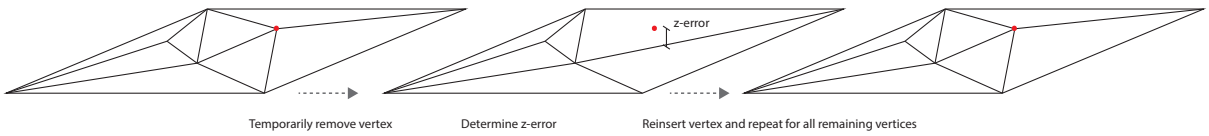


Figure 2.12: How z-error is determined for drop-heuristic decimation in a triangulation. Figure drawn in perspective viewing a triangulation at an angle from above.

Decimation can also be done by means of *edge contraction*, which is a method that uses a metric based on the sum of the quadric error of two vertices. The two vertices with the overall lowest quadric error sum between them are subsequently contracted into a single vertex along their connecting edge. The quadric error is determined based on a matrix calculation using the squared distance of a vertex v to a a plane p, about which more details can be found in Garland and Heckbert [1997b, p. 4]. Garland and Heckbert [1997b] also show how this metric can be applied to perform mesh simplification by means of edge contraction to achieve good results.

By using the quadric error edge contraction method, vertices are physically moved to find the optimal position along the edge connecting the two vertices with the lowest combined error metric. The result of this is a more fine-tuned ϵ as more granular adjustments can be made to the triangulation. Despite this, this method is not used as one of the possible simplification algorithms within my research, because the computational complexity of calculating the sum of quadric errors for each vertex pair is larger than calculating a vertical error delta. Nevertheless the quadric error edge contraction algorithm is a method which should be applicable to simplification in a streaming pipeline and should be considered for future work within this research topic, as further discussed in Section 5.4.

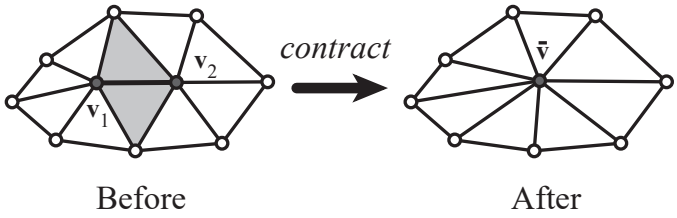
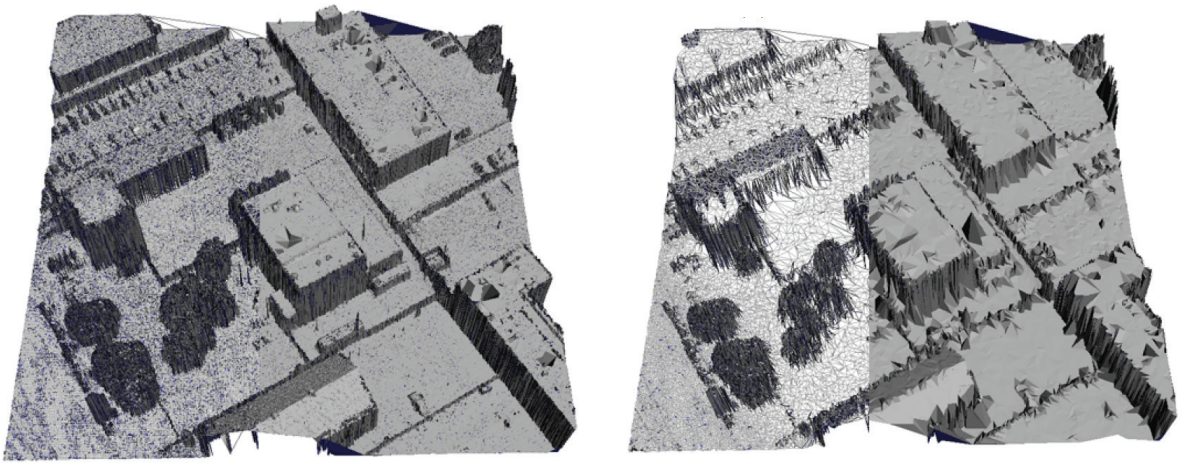


Figure 2.13: An example of how edge contraction shifts and merges vertices along the connecting edge. The highlighted edge is contracted to a single point. The shaded triangles are removed. Figure from Garland and Heckbert [1997a, p. 2].

An extension of this quadric error method is introduced by Constantin et al. [2010] who allow for extra limitations to be provided. Such limitations are: only simplifying points with a certain classification, using object ID's to determine what should be simplified, or using building boundaries as constraints. This approach is particularly interesting because it provides a way to apply local simplification in a streaming pipeline, allowing for reductions in the number of vertices up to 95% while still maintaining the structure of the TIN, as seen in Figure 2.14.



(a) Original mesh (995k vertices)

(b) Simplified mesh (49k vertices)

Figure 2.14: Streaming simplification of a TIN which shows how features are maintained. The left half of each figure only shows triangles, the right half also shows their surfaces. Figures from Constantin et al. [2010, p. 154].

One of the innovative ideas introduced by Constantin et al. [2010] is to adjust the target error threshold during processing. This is done based on the error of the portions of the triangulation which have already been completed. By doing so, corrections can be made for the global error metric by monitoring how the local error metric is developing over time. If the error of the already finalized sections of the triangulation is higher than the specified global threshold, the local threshold for the remaining sections of the triangulation is lowered. The same works in the opposite way, where the local threshold can be

increased if the error of the finalized sections is lower than the global threshold.

Constantin et al. [2010] state a performance that scales linearly with the input size for an approximate throughput of 50,000 vertices per second Constantin et al. [2010, p. 155]. This is achieved by using a buffer of a set number of vertices within which edge contraction with quadric error measurement is applied as simplification algorithm. The buffer is filled by new triangles coming in from the streaming pipeline and cleared as triangles are finalized following simplification.

The largest dataset simplified using this approach is 2.3GB, containing 65 million vertices, similar to 20% of a single AHN3 tile. Constantin et al. [2010] state they are able to simplify this dataset in 19 minutes keeping 25% of all the points in the input set. An error metric is not provided for this specific dataset. Another, smaller, dataset is run using the same methodology to achieve an error-per-vertex (EPV) (which is the same as RMSE) of 309 for a dataset of 995,000 vertices and 0.15 for a dataset of 200,000 vertices. Why there is such a large difference between the EPV of the 995,000 vertices dataset compared to the 200,000 vertices dataset is not further explained in their research.

2.4.2 Refinement Principle

The second main method for TIN simplification is refinement. This type of simplification is done by starting with a very basic (e.g. four corners of bounding box) triangulation and subsequently checking which point has the largest error and adding that point to the triangulation. Error is defined by linearly interpolating the value of the point on the triangulation and comparing that to the actual z-value, as shown in Figure 2.15. In each iteration the point with the largest error is added to the triangulation until a certain user-specified threshold for minimum accuracy of the triangulation is reached.

Refinement can be implemented as a greedy algorithm, which means that the algorithm makes the best local decision using the information at hand. This can result in a locally optimal solution that adheres to the user-specified maximum error threshold. However, this means that the threshold is not guaranteed globally and the global threshold is likely to be surpassed [Ledoux et al., 2021a, p. 84-85]. Two different approaches to a refinement algorithm are used for this thesis. The one uses the error of the next point in the stream, inserting it immediately if the error is above the threshold, while the other approach uses the point with the largest error available in a larger subset of points. This difference, and the difference in the outcome, is highlighted in Section 3.2.3.1 and Section 3.2.3.2.

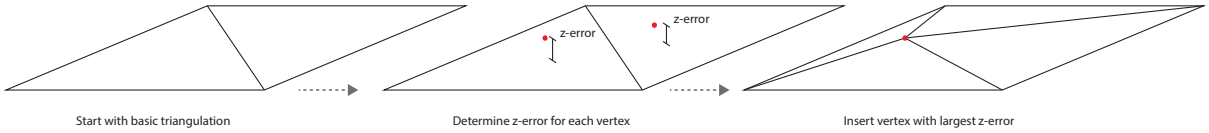
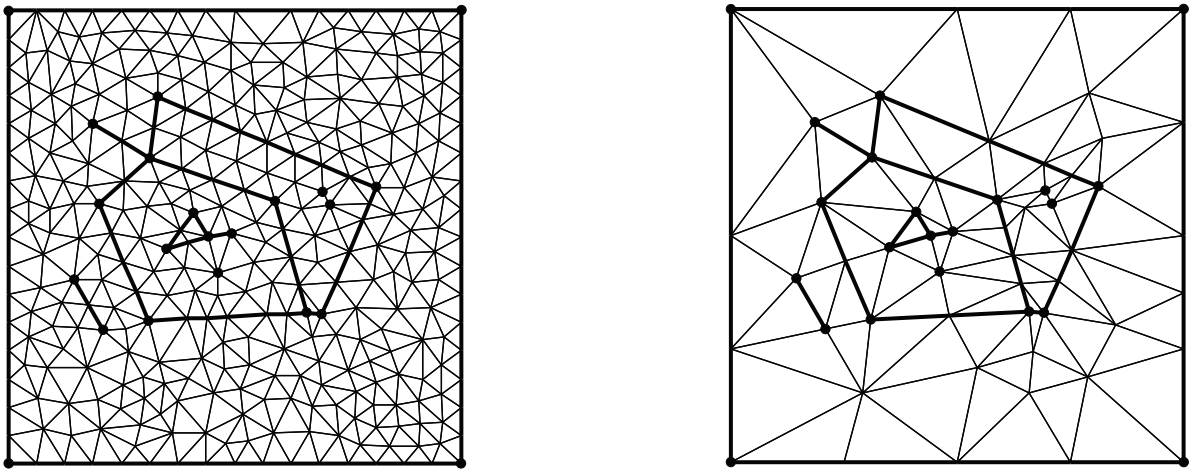


Figure 2.15: How z-error is determined for refinement methods and which vertex is subsequently inserted into the triangulation. Figure drawn in perspective viewing a triangulation at an angle from above.

The origin of usable refinement algorithms lies with Ruppert [1995], who is able to produce meshes without small angles and with control over the density of triangles. This research focuses mainly on the creation of constrained Delaunay triangulations and simplifies triangulations by specifying a minimum angle for triangles. Defining a minimum angle for each triangle results in a sparser triangulation due to enforcing the creation of larger triangles through this metric. What effect this constraint has on a triangulation can be seen in Figure 2.16.

The refinement method defined by Garland and Heckbert [1997a] is one of the more practical and efficient methods. It makes use of greedy insertion, meaning that it will insert one or more points into the triangulation on each pass. Furthermore, this method makes use of a max heap to store the vertex errors. A max heap is an efficient data structure for storing ordered values and removing/inserting points. The



(a) Triangulation with a minimum angle of 22.5°

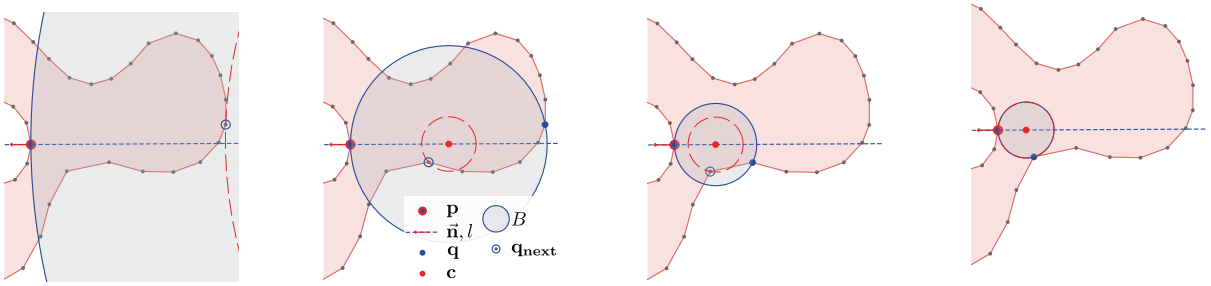
(b) Triangulation with a minimum angle of 20°

Figure 2.16: How a minimum angle requirement affects the degree of simplification in a constrained Delaunay triangulation. Figure from Ruppert [1995, p. 3].

advantage of the method defined by Garland and Heckbert [1997a] is that it only stores the worst error vertex per triangle and only recalculates the errors of vertices that are affected by an insertion. This method is practical if there is knowledge on which points are contained in a specific triangle because this allows for recalculating only the vertices affected by the insertion of another vertex. However, within the triangulation software used for this thesis (Startin) no information is provided on which points are contained within a specific triangle. Therefore it is not practical to use the method defined by Garland and Heckbert [1997a] as it would rely on doing expensive point-in-triangle checks.

2.4.3 Medial Axis Transform

The last main simplification method is based on using a medial axis transform (MAT) combined with the local feature size (LFS) to simplify massive point clouds. A MAT is a 3D surface representation of a point cloud that approximates the surface created by the points. It is created using the medial axis shrinking ball (MASB) algorithm, which is seen in Figure 2.17. Using a sphere of decreasing radius this algorithm is able to determine when it is 'inside' the point cloud by checking whether the sphere is contained in the point cloud. The interior and exterior MAT represent the interior and exterior surface of the point cloud. It should be noted that it is generally necessary to have a buffer of points around the target area to create an accurate MAT. This will not be possible within the streaming geometries pipeline and will thus likely have a negative effect on the results of this method.



(a) Initial ball

(b) Second iteration

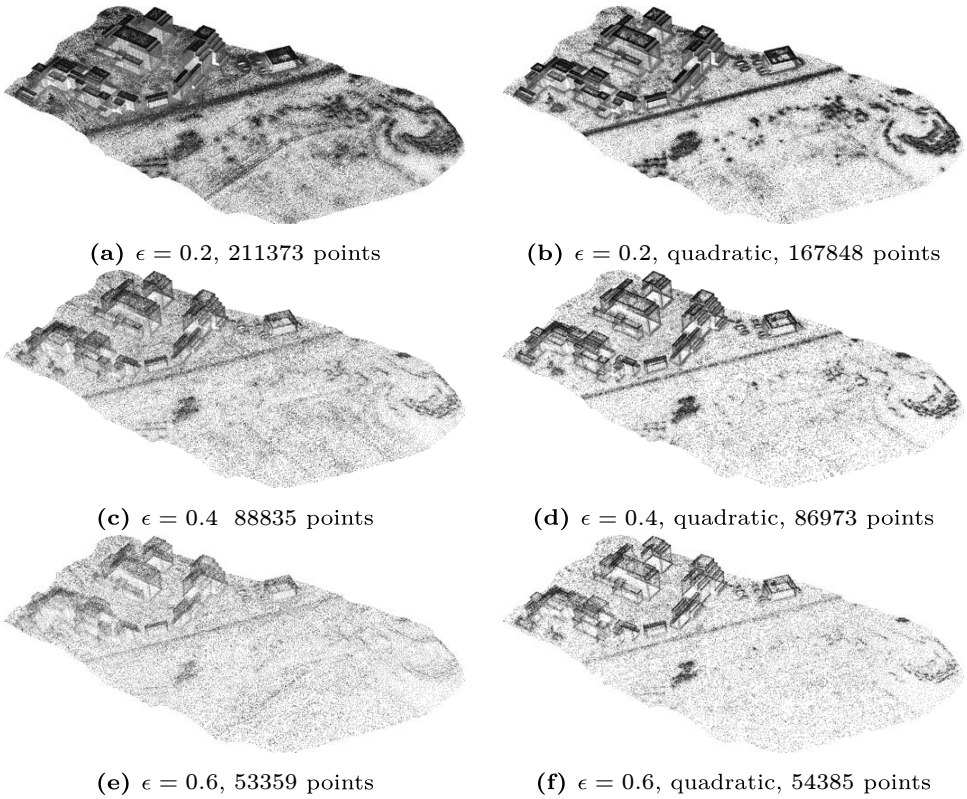
(c) Third iteration

(d) Fourth iteration yields an empty ball

Figure 2.17: How MASB creates the interior surface of a dataset. Figure from Peters [2018a, p. 37].

The LFS is a measure introduced by Amenta and Choi [2008] which represents the shortest distance between a surface point and an MAT. Using LFS allows for determining whether a point is geometrically significant by identifying if the curvature near the point is high [Peters, 2018a, p. 89]. A high curvature indicates that the point is most likely significant as it lies near a sharp change in the surface structure. These points should be kept as they define edges of features, whereas points with lower significance (curvature) can be discarded.

The method avoids using a KD-tree for simplification by using a three-dimensional regular grid to approximate the thinning factor and subsequently randomly thinning a grid cell Peters [2018a, p. 90]. Using the three-dimensional regular grid ensures that the simplification algorithm is able to run in linear time. Furthermore, Peters [2018a] provides a number of parameters that can be set by the user, making it challenging to optimize the parameters without sufficient knowledge of the inner workings. However, if a near-optimal set of parameters is achieved the results are promising, as can be seen in Figure 2.18.



(a) $\epsilon = 0.2$, 211373 points

(b) $\epsilon = 0.2$, quadratic, 167848 points

(c) $\epsilon = 0.4$ 88835 points

(d) $\epsilon = 0.4$, quadratic, 86973 points

(e) $\epsilon = 0.6$, 53359 points

(f) $\epsilon = 0.6$, quadratic, 54385 points

Figure 2.18: Results of MAT point cloud simplification for various ϵ. Figure from Peters [2018a, p. 92].

3 Methodology

This chapter discusses the methodology developed to construct and simplify massive Delaunay triangulations in a single streaming pipeline. Simplification is added to the streaming geometries pipeline sst, as described in Section 3.1. Depending on the simplification algorithm, the position of the algorithm within the pipeline can change between one of the three possible positions. Different implementations of the simplification types discussed in Section 2.4 are explained in the following sections, as well as reasons why the simplification method may work best in a streaming pipeline. Furthermore, there is one control method called randomized thinning which is used as a comparison.

Section 3.2 goes more into depth about different adaptations that have been made to the simplification methods to use them within a streaming pipeline. Most notably, a combined methodology between refinement and decimation is introduced and some initial results are presented.

Lastly, Section 3.3 discusses which parameters are relevant to judge the performance of the simplification methods within the streaming pipeline. These evaluation criteria are essential in understanding how each method performs for different parameters. Therefore, four criteria are identified which will provide insight to the results of each simplification method.

3.1 Integrating Simplification in a Streaming Pipeline

In the following sections the design choices for implementing simplification into sst are explained, as well as more in-depth information on the placement of the simplification methods within the streaming pipeline. The different types of simplification are tested for their speed, accuracy, and memory usage. These results are important because the algorithms need to be capable of processing massive TINs, thus a trade-off between these three factors is required. Based on the results from these tests, variations of the simplification methods are created that solve different problems such as; keeping too many triangles, and taking too long to compute the simplified triangulation.

3.1.1 Architecture of Simplification within the Streaming Geometries Pipeline

Simplification can be implemented into the streaming geometries pipeline by adding a simplification algorithm within the finalizer or the triangulator. The advantage of implementing simplification within the existing finalizer or triangulation is that their functionality is expanded. However, when assessing this design choice from the streaming geometries principle there is a better option; making the simplification method its own module in the streaming pipeline.

Making the simplification method a separate module in the streaming pipeline will allow for the use of any programming language in creating the module. The major advantage of the streaming pipeline is that it relies only on a standard input and standard output. By using this modular approach the simplification methods can be written in any programming language and still work flawlessly with the other components in the pipeline. In turn, this approach creates a more sustainable ecosystem of components and allows for a lower barrier to entry if other researchers want to create their own modules.

Furthermore, in the future this design choice may lead to the creation of other modules that provide separate functionalities. An example of another useful module is a streaming visualization module, such as the one discussed in Section 4.1.4. Other possible modules that come to mind are ones that

follow the ideas of Isenburg et al. [2006d], in which contour lines are created in a streaming pipeline, or a DEM is generated. Anyone, in any programming language, could choose to create a module using the output from the finalizer or triangulator that turns their result into valid contour lines. Therefore, the simplification implementations serve a good example project and are created as an *sstmod*(ule) to provide an example for future usage.

3.1.2 Possible Placement of Simplification in the Streaming Pipeline

The decision to make the simplification methods modular leads to another decision, namely where to place these modules within the streaming pipeline. There are three main positions where simplification could be implemented: A, B, or C in Figure 3.1.

Figure 3.1: Possible placement positions for a simplification module within the *sst* pipeline.

The advantage of position A is that it simplifies the input data, ensuring that the finalizer and triangulator have less points to process which reduces the time taken by these steps. The disadvantage is that not all of the methods can work with raw point cloud data. Furthermore, methods in position A rely on the dataset having extremely high spatial coherence, as points are passed to the simplification method entirely sequentially. A possible consequence of this decision can be seen in Figure 3.13 and is further discussed in Section 3.2.5.

For position B, the advantage is that the finalizer has already categorized these points into cells and releases them cell-by-cell. Doing so ensures that there is always a similar number of points released to the simplification module, as well as that they are highly spatial coherent. This correlates with the work of Funke et al. [2019] in spreading the workload evenly across multiple cores. There is a possibility that simplification within each cell may lead to artefacts on the borders of the cells due to these being sharp, strict, borders. Whether artefacts are actually introduced is further analyzed in Section 3.4.

Position C has the advantage that it can work with a finalized triangulation, which means that it prevents possible border artefacts from occurring. The disadvantage of position C is that the triangulator will need to triangulate all points in the dataset prior to passing this information to the simplifier, increasing its computation time. Furthermore, the simplification module will also have to deal with more data compared to position B, because it relies on using the triangles released by the triangulator as opposed to only vertices as released from the finalizer.

Based on the above advantages and disadvantages, different simplification algorithms are implemented at different positions. The effects of these choices is further analyzed in Chapter 4.

3.2 Use of Simplification Methods in the Streaming Geometries Pipeline

The following sections discuss the various simplification methods that are implemented. Each section shows what the position is of the module within the streaming pipeline and shows the input and output data processed by the module.

3.2.1 Randomized Thinning

Randomized thinning is used as a control method to compare with the results of other simplification modules. Randomized thinning is the most basic form of simplification, thus providing a clear baseline when a method performs better than random. Ideally this method is implemented at module position A, as this reduces the workload for the finalizer. However, doing so requires rewriting the finalizer to accept data from a prior module, which it is currently not capable of. Therefore, to ensure modularity is maintained, randomized thinning is implemented at position B, shown in Figure 3.2.

Figure 3.2: Randomized thinning of the input dataset as simplification method implemented at position B in the *sst* pipeline.

The randomized thinning method relies on randomly allowing one in every n points through the streaming pipeline and on to the next application. This is done by using a pseudo-random number generator that generates numbers between 1 and n. When the result of the random generator is $n/2$, the value is allowed to pass through to the next application. In further sections, randomized thinning is referred to as rand + n, e.g. *rand100* if one in every 100 points is passed on.

3.2.2 Drop-Heuristic

Drop-heuristic is a decimation-type simplification which relies on removing points that are below a certain accuracy threshold from a triangulation, as described in Section 2.4.1. The drop-heuristic method is implemented at position C in the streaming pipeline (see Figure 3.3). The reason for this is that it is desirable to use the same triangulation twice, i.e. in the triangulation and for drop-heuristic. To do so, the triangulator sends its entire triangulation to the drop-heuristic module to reuse for simplification. Doing so the other way around, with drop-heuristic sending a full triangulation to the triangulator, is not possible due to requiring changes to the triangulator.

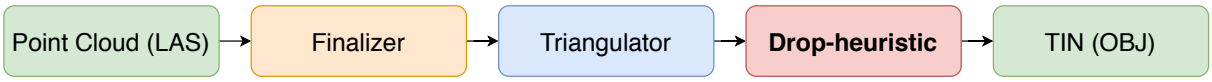


Figure 3.3: Drop-Heuristic implemented at position C in the *sst* pipeline.

Figure 3.4 shows what the input for drop-heuristic simplification looks like. This input is received line-by-line from the triangulator and subsequently simplified using the drop-heuristic method. Line 1 is the line that provides the bounding box of the dataset, which is ignored for drop-heuristic simplification. Lines 2-6 show how vertices are received by drop-heuristic in x, y, z order. Line 7 indicates that the vertex with id *1* has all of its neighboring vertices already present and can therefore be finalized. To ensure that this simplification module is able to rebuild the same triangulation as provided by the triangulator, line 7 also contains the vertex id's of the neighbors in counter-clockwise order. How the triangulation is rebuilt using the provided information can be seen in Figure 3.5.

Using the information seen in Figure 3.4, the exact same triangulation as created by the triangulator is rebuilt inside the drop-heuristic method. Doing so ensures that the triangulation is built in the same way and no recalculation is required. Furthermore, this ensures that it is clear when a point can be finalized, as it should have all its neighbors loaded before it can be finalized. As can be seen in Figure 3.5, it is possible for vertex 1 to be finalized, while not all of its neighbors are finalized. In the example, vertex 1, 2, and 5 are finalized, but 3 and 4 are not. This means that vertex 1 is not ready to be simplified yet and can be excluded from the simplification for now.

```
1     b 84600.000 447000.000 84699.998 447099.999
2     v 84662.229 447029.124 0.745
3     v 84662.238 447029.101 0.745
4     v 84662.315 447029.061 0.759
5     v 84662.222 447029.331 0.758
6     v 84661.998 447029.189 0.722
7     x 1 [2, 3, 4, 5]
8     v 84656.722 447016.696 3.582
9     v 84656.805 447016.542 0.683
10    v 84656.812 447016.733 0.704
11    ...
```

Figure 3.4: Output example from the Triangulator showing how edges and vertices can be rebuilt for use in drop-heuristic decimation.

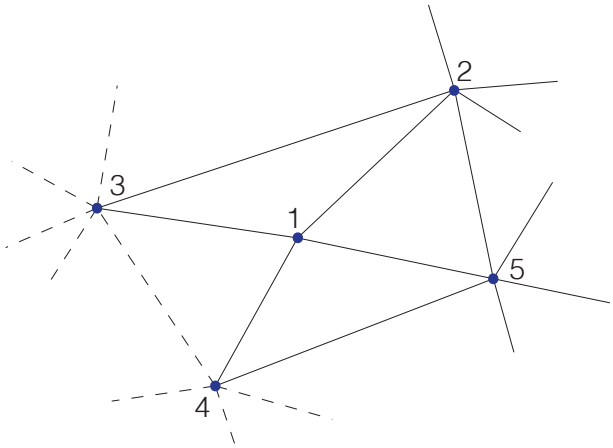


Figure 3.5: Example of what the data from Figure 3.4 may look like after more points are finalized. Dotted lines represent non-existent edges to unfinalized neighbors.

Simplification with drop-heuristic decimation works by waiting for a number of points (e.g. 5000) to be received as input before attempting to simplify all points that are now held in memory. Due to the requirement of vertices and their neighbors needing to be loaded into memory it is possible that none, or very few, points are simplified in a single pass. In that case, another set of the same number of points are received and simplification is re-attempted. This process is repeated until enough vertices are available and ready to be finalized to perform the simplification process and the simplified points can be released to the standard output. All vertices that are not ready to be finalized are kept in memory to be finalized at a later moment, when they adhere to the requirements to be finalized. Algorithm 3.1 explains the drop-heuristic simplification as it is implemented in pseudo-code to provide a better overview of each step.

Choosing different recalculation intervals has an effect on both the accuracy and the computation time. A larger recalculation interval will lead to less computation time as the errors are recalculated less frequently. However, because multiple vertices are removed at a time it may lead to relevant vertices being removed as well. Relevant vertices may be removed despite ordering these vertices for the largest error. As seen in Figure 3.6, it may occur that the ten vertices that are scheduled for removal are all within the same area. If this area represents a building, then all vertices that represent it are removed, resulting in a large error in the final TIN compared to the original dataset. This can be mitigated by double-checking whether a vertex is valid for removal prior to actually removing it, but this would add extra computational complexity to the simplification method. The naming used for drop heuristic decimation is decim + the recalculation interval, e.g. decim10 for a recalculation interval of once every ten insertions.

Algorithm 3.1: Drop-heuristic decimation

```
1  foreach line from stdin do
2      lineIdentifier = first char of line from stdin
3      data = rest of chars of line from stdin
4      if lineIdentifier == "v" then
           // data contains vertex x, y, z
5          insert data into triangulation as new vertex
6      else if lineIdentifier == "x" then
           // data contains center vertex id followed by list of neighbor vertex ids
7          define a new star in triangulation (center vertex + neighbors)
8      end
9      if length(verticesInserted) % processingInterval == 0 then
10         for all finalized vertices do
11             zError = abs(trueZ - interpolatedZ)
12             push zError to heap
13         end
14         while heap contains an element do
15             smallestErrorVertex = pop from heap
16             if zError of smallestErrorVertex < errorThreshold then
17                 remove smallestErrorVertex from triangulation
18             end
19             if length(heap) % recalculationInterval == 0 then
20                 recalculate zError for all vertices
21             end
22         end
23         Release all points from triangulation to standard output
24         Clear vertices
25     end
26 end
```

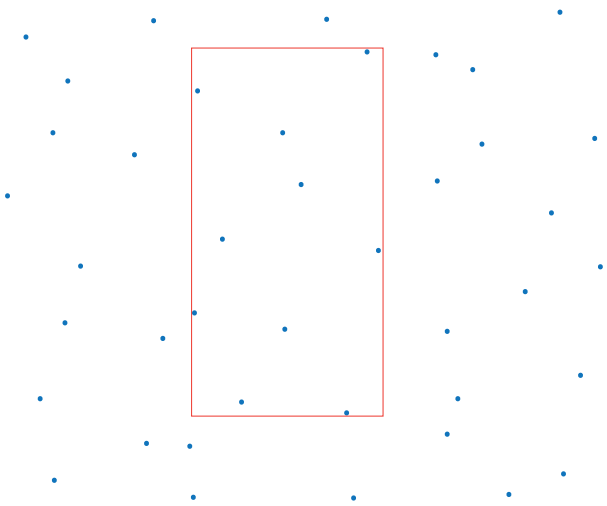


Figure 3.6: How drop decimation as used can lead to removal of relevant vertices. When removing ten vertices at a time it may occur that all vertices within the red building footprint are removed. This results in a large error between the TIN and the original dataset.

3.2.3 Refinement

Refinement works almost opposite from drop-heuristic decimation; the algorithm looks for the vertex with the most error and adds this vertex to the triangulation. Repeating this process until the vertex with maximum error is below a user-specified threshold, thus *refining* the triangulation on a point-by-point basis. Therefore, the outcome of both methods is similar even though the algorithms are totally different. However, with refinement, there are some decisions that can be made to implement the method in a number of different ways. In the following sections two different implementation choices are discussed; *greedy* refinement and *first-come-first-serve* refinement.

Figure 3.7: All types of refinement are implemented at position B in the *sst* pipeline, as shown.

The position of refinement in the pipeline is at position B, as seen in Figure 3.7. The reason for choosing this position for refinement is that the refinement methods use the cells released by the finalizer as blocks within which simplification is performed. Performing simplification within the released cells is different from decimation, which relies on a number of points to accumulate prior to simplifying. Using the released cells to simplify the data is an advantage because there is a very clear grouping of data that is to be simplified. The disadvantage is that there is a possibility of introducing border artefacts around tile borders, further discussed in Section 3.4.

With the placement of the refinement simplification algorithms in position B it means that a triangulation will be created by the refinement algorithm prior to the triangulator. Doing so means that the triangulation is re-created in the triangulator using the vertices from the simplified triangulation, instead of keeping the triangulation that has already been created by the refinement algorithm. Therefore, an optimization that can be made is to ensure the output of refinement in position B can be directly stored to an output OBJ file. Doing so provides two distinct advantages: less processing time needed because one step is removed from the pipeline, and the output triangulation is guaranteed to be the same as for which the error is measured. The latter cannot be guaranteed when keeping the triangulator as the triangulator will re-triangulate the vertices, which may result in a different outcome. Despite being more efficient this option is not used for any of the simplification algorithms at this time. However, it is a relevant option further discussed as possible future work in Section 5.4.

The refinement algorithms receive information from the finalizer as shown in Figure 3.8. Lines 1-4 denote background information about the dataset that is to follow; number of vertices (n), number of cells in x and y direction (c), width and height of each cell (s), and the bounding box (b). Of these lines, the bounding box and the width and height of each cell are used by the refinement algorithms. These are used to be able to determine artificial corner points necessary to initialize a triangulation, further discussed in Section 3.2.3.1. All these lines are necessary at the top of the file for the triangulator to function, thus they are also sent to the triangulator when they are received.

The lines that follow (line 5 onwards) contain all the vertices that are being sprinkled into the triangulation which are left untouched by the refinement algorithms. These sprinkle vertices are used to prevent the triangulation from creating slivers. When the tag on line 9 (# endsprinkle) is received, the refinement algorithm knows that all sprinkle vertices have passed and it should simplify all the following vertices. Every following line starting with v denotes a new vertex. Line 14 shows a finalization tag for cell x=1, y=0, which means that all vertices within that cell have been released in the stream and the cell can be simplified.

Data released by the refinement module is sent in the same format as it is received, this is done to ensure that no modifications are necessary to the triangulator. There is a discontinuity here where the refinement algorithm will report the original number of vertices to the triangulator, but sends less due to the simplification. This has no further consequences besides marginally increasing the memory usage of the triangulator.

```
1   n 212550
2   c 2
3   s 50
4   b 84600.000 447000.000 84699.998 447099.999
5   v 84693.724 447051.214 1.410
6   v 84628.038 447085.693 9.739
7   v 84684.830 447073.781 0.877
8   ...
9   # endsprinkle
10  v 84650.190 447038.811 7.305
11  v 84650.147 447038.293 7.528
12  v 84650.111 447037.837 7.735
13  ...
14  x 1 0
15  v 84699.781 447050.264 1.284
16  v 84699.508 447050.050 1.307
17  v 84699.989 447050.783 1.277
18  ...
```

Figure 3.8: Sample input for all refinement algorithms as received from the finalizer.

3.2.3.1 Greedy Refinement

Greedy refinement is a methodology that relies on storing all points within a cell received from the standard input and then creating a simplified triangulation point-by-point. This method is created by Garland and Heckbert [1997b] and is named greedy because it inserts one or more vertices on every pass. However, it should be noted that greedy refinement still uses a global measure to determine which vertex is the best candidate to be inserted. This differs from other greedy refinement techniques that use a similar method to Section 3.2.3.2.

To perform greedy refinement, an initial triangulation is created using four corner points to ensure all subsequent points are within the convex hull of this triangulation and can be interpolated. The X and Y position of these corner points is calculated by using the corners of the cell that is being finalized. The z-value of these corner points is determined by taking an average of the ten nearest vertices to the corner. This allows for a relatively accurate elevation value for these corner points to ensure they are representative for the triangulation that follows. Using this initial triangulation, the error of each remaining point is calculated to improve the accuracy of the triangulation, after which the point with the largest error is inserted into the triangulation. The implementation of this algorithm is seen in pseudo-code in Algorithm 3.2.

Ideally, the errors of each remaining vertex are recalculated every time after a new vertex has been added to the triangulation. This is because with the adding of each vertex the triangulation changes and thus the error of all remaining vertices. However, due to the computational cost of recalculating the errors, it is not feasible to do so while still simplifying the triangulation in a reasonable amount of time. One approach is to recalculate the error every n insertions, i.e. insert 10 vertices before recalculating the errors. The disadvantage of this is that at the start of the refinement process the odds are very high that the n vertices with the highest error are located very close to each other. This is because the triangulation is still flat at this moment in time, thus all vertices on the tallest object will have the largest error. This results in the creation of numerous small triangles which do not affect the accuracy of the result, but add unnecessary triangles to the triangulation, as seen in Figure 3.9.

To avoid this, the errors can be recalculated incrementally. What this means is that at the start of the refinement process, when the triangulation is flat, the errors are recalculated after every insertion. However, as the triangulation more accurately approximates the true values of the vertices, the probability that the vertices with the worst errors are all located next to one another decreases. This means that if

Algorithm 3.2: Greedy refinement

```
1  foreach line from stdin do
2      lineIdentifier = first char of line from stdin
3      data = rest of chars of line from stdin
4      if lineIdentifier == "v" then
           // data contains vertex x, y, z
5          insert data into triangulation as new vertex
6      else if lineIdentifier == "x" then
7          if length(vertices) > 0 then
8              foreach vertex in vertices do
9                  push zError to heap
10             end
11             while heap contains an element do
12                 largestErrorVertex = pop from heap
13                 if zError of largestErrorVertex > errorThreshold then
14                     insert largestErrorVertex into triangulation
15                 else
16                     break loop
17                 end
18                 if length(vertices) % floor(recalculationInterval) == 0 then
19                     recalculate errors of vertices still in heap
20                     increment recalculationInterval by step size (0.25)
21                 end
22             end
23             Remove artificial corner points from triangulation
24             Release all vertices from triangulation to standard output
25         end
26     end
27 end
```

more vertices are inserted at once there is a lower chance that it will lead to the creation of small unnecessary triangles. Therefore, the interval at which recalculation is done is slowly increased over time to reduce the number of times this costly operation has to be run.

The identifier for greedy refinement is therefore based on the step size used to increase the recalculation interval. For a step size of 0.25, this will increase the recalculation interval by one every four insertions which means the identifier is Greedy025. For a step size of 1, this will increase the recalculation interval by one every insertion which means the identifier is Greedy1.

3.2.3.2 First-Come-First-Serve Refinement

First-Come-First-Serve (FCFS) refinement relies on inserting vertices as soon as possible. This is done by inserting a vertex into the triangulation at the moment it is received in the case that its error is above the user-specified threshold. By doing so, there is no longer a waiting time between receiving the vertex and determining whether it should be added to the triangulation. Furthermore, it prevents the computationally heavy task of determining the error for each vertex.

This results in a very fast simplification method, as it removes the need to recalculate the errors of each vertex to determine whether they should be inserted into the triangulation. Pseudocode showing how FCFS refinement is implemented can be seen in Algorithm 3.3. What can be noticed is that there are two moments at which points can be inserted into the triangulation; upon being received and when the cell is finalized. The reasoning behind this is that if the second processing stage upon finalization is

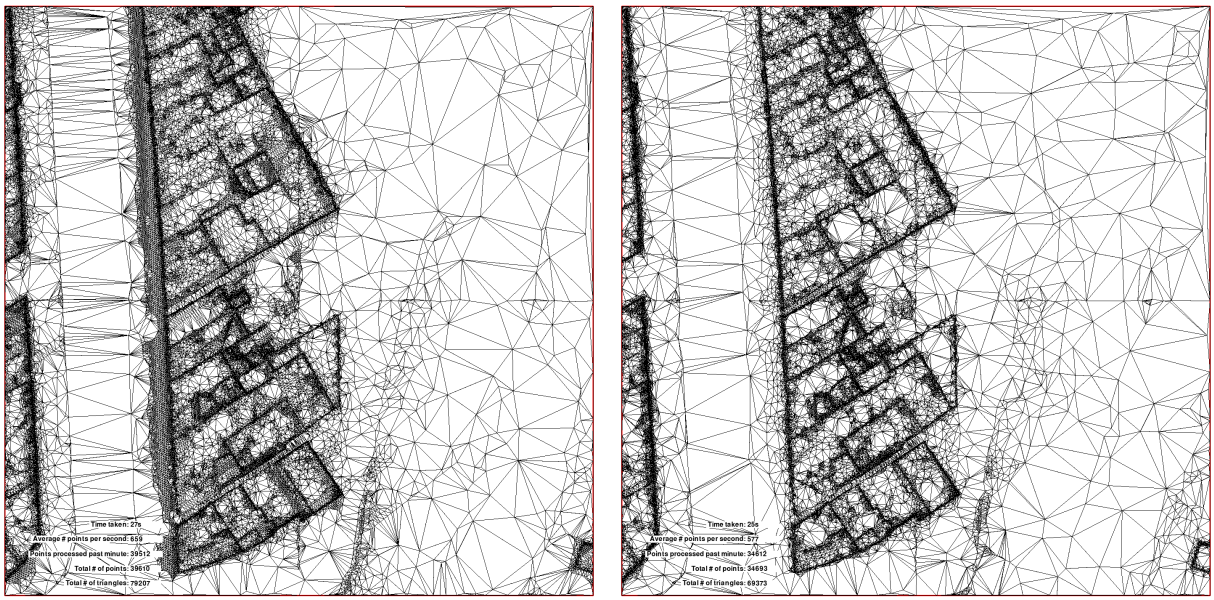
not added, there are too many small triangles around sharp inclines. The difference between this can clearly be seen in Figure 3.9. The example here shows what the triangulation looks like when vertices are always inserted directly and no secondary processing step is added.

The goal for the coarse threshold, seen on line 7, is to create an initial approximation of the triangulation. By adding only the vertices with a very large error ensures that the triangulation follows the shape of the most distinct objects. This means that for the loop starting from line 13 the errors of these vertices are much lower. The result of this is that only the smaller features are added to the triangulation.

FCFS as used to create the results discussed in Section 4.4 are all created using a primary threshold of 2m and a secondary threshold of 0.2m. Therefore whenever FCFS is mentioned these threshold can be assumed.

Algorithm 3.3: First-Come-First-Serve Refinement

```
1  foreach line from stdin do
2      lineIdentifier = first char of line from stdin
3      data = rest of chars of line from stdin
4      vertices = empty list
5      if lineIdentifier == "v" then
           // data contains vertex x, y, z
6          zError = abs(trueZ - interpolatedZ)
           // coarseThreshold is around 2 meters
7          if zError > coarseThreshold then
8              insert vertex into triangulation
9          else
10             append vertex to vertices
11         end
12     else if lineIdentifier == "x" then
13         foreach vertex in vertices do
14             zError = abs(trueZ - interpolatedZ)
               // fineThreshold is desired final accuracy
15             if zError > fineThreshold then
16                 insert vertex into triangulation
17             end
18         end
19         Release all vertices from triangulation to standard output
20     end
21 end
```



(a) FCFS refinement with a single insertion moment with 0.2m threshold.

(b) FCFS refinement with two insertion moments, the first with 2m threshold, the second with 0.2m.

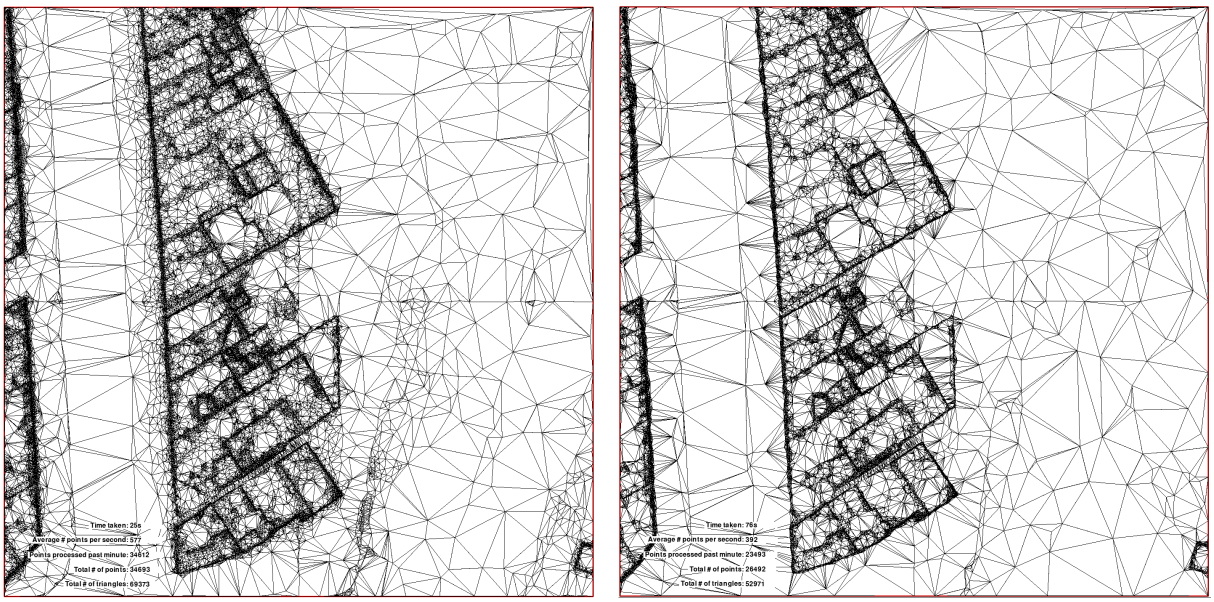
Figure 3.9: Comparison of two First-Come-First-Serve refinement techniques. Clear improvement in the number of small triangles between (a) and (b).

3.2.4 Combining First-Come-First-Serve Refinement with Drop-Heuristic Decimation

Combining FCFS refinement with drop-heuristic decimation is implemented in the same place within the streaming pipeline as refinement, as seen in Figure 3.10. This method introduces an extra step to FCFS refinement method by adding drop-heuristic decimation at the end. This ensures that the triangulation created by FCFS, which contains numerous small triangles, is cleaned up by the drop-heuristic decimation.

Figure 3.10: FCFS + Decimation simplification implemented at position B in the *sst* pipeline.

The artefact introduced by FCFS refinement is that multiple nearby points are inserted because each point has an error that is above the threshold at the moment they are checked. Adding a decimation step after the secondary insertion loop ensures that this artefact is removed from all present locations by double-checking whether a vertex is necessary to maintain the specified error threshold. The reason this approach is chosen is because decimation on its own runs quite slowly due to the sheer number of vertices that need to be recalculated. By initially creating the triangulation using the FCFS refinement method, this results in a reduction of the number of vertices that need to be calculated by the decimation method. A comparison between the greedy refinement method and this hybrid method can be seen in Figure 3.11.



(a) FCFS refinement with two insertion moments, the first with 2m threshold, the second with 0.2m.

(b) FCFS refinement with two insertion moments and a decimation step as final filtering.

Figure 3.11: Comparison of FCFS refinement against FCFS with decimation. Visible improvement to the number of small triangles can be seen, especially around steep inclines.

This simplification algorithm will be identified as FCFS+Decim(recalculation interval). For example, for a recalculation interval of once every 100 points the identifier is FCFS+Decim100. This is in line with the previously mentioned identifiers for each simplification algorithm.

3.2.5 Medial Axis Transform Simplification

Figure 3.12: MAT simplification implemented at position B in the *sst* pipeline.

The MAT method uses the code of Peters [2018a] which is provided as open-source online under the MIT license, allowing for very flexible usage for this simplification method [Peters, 2018b]. MAT simplification is implemented at position B in the streaming pipeline, shown in Figure 3.12. Implementation of this method relies on using the code provided by Peters [2018a] and adjusting usage for application in the streaming pipeline. This is initially done by combining the three separate stages of the program into a single file, allowing for easier piping of data in and out of the MAT simplification. The calculation steps taken in the various functions within the simplification method are seen as a black box as they are out of scope for this research.

Algorithm 3.4: MAT Simplification

```
1  foreach line from stdin do
2      lineIdentifier = first char of line from stdin
3      data = rest of chars of line from stdin
4      vertices = empty list
5      if lineIdentifier == "v" then
6          insert vertex into array of points
7      else if lineIdentifier == "x" then
8          remove duplicate points
9          compute normals of all vertices
10         compute the medial axis shrinking ball (MASB) of the points
11         simplify the local feature size (LFS)
12         release simplified points to standard output
13     end
14 end
```

For this method, it is possible to place it in position A due to the ability to simplify a raw point cloud. However, the decision is made for position B to ensure that all cells processed by the simplification are the same size. This is relevant for the MAT simplification method because otherwise the resulting density of the point cloud varies, as can be seen in an example in Figure 3.13. This example shows the effects of processing the point cloud while relying entirely on the inherent spatial coherence of the dataset. Furthermore, this means the same approach is used as for other methods implemented at position B, such that vertices are stored until a finalization tag is read, after which all vertices stored at that moment are simplified. This method relies entirely on the implementation created by Peters [2018a]. For this reason the inner workings of the algorithm are more unknown compared to the other simplification methods. The general overview of how the algorithm works can be seen in Algorithm 3.4.

MAT simplification is further identified as MAT(epsilon), where the epsilon is the error threshold specified to the MAT simplification algorithm as designed by Peters [2018a]. For example, MAT simplification with an epsilon of 0.4 is identified at MAT04.

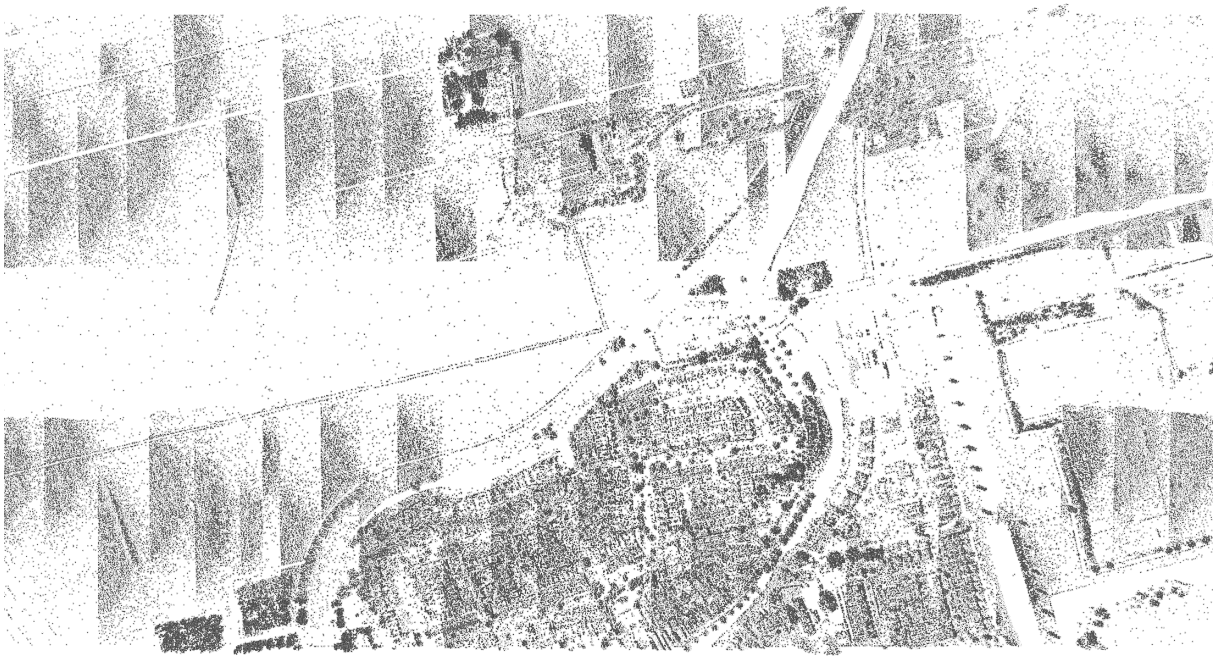


Figure 3.13: Variation in output point cloud density when simplifying using MAT simplification based on n points per processing loop. This varying density results in a TIN with varying density, something that is not desirable.

3.3 Evaluation Criteria: Parameters that Determine the Optimal Simplification Method

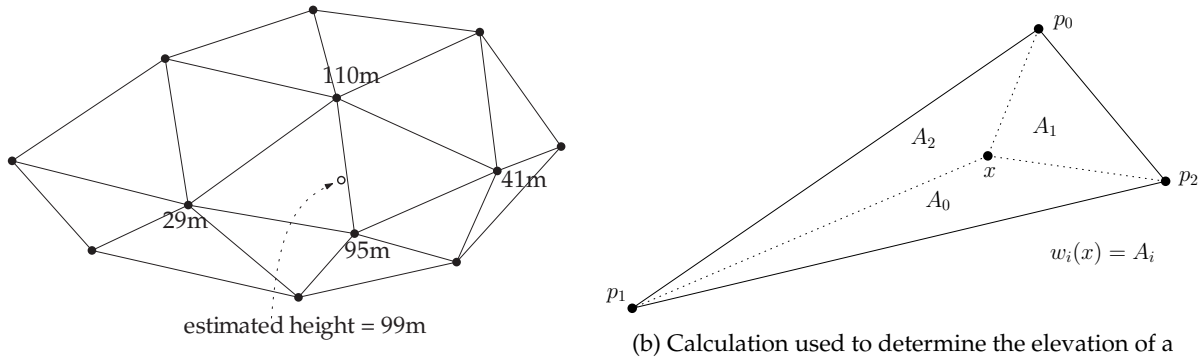
The following sections explain which evaluation parameters are necessary to determine the success of each simplification method. Despite accuracy being an important parameter for simplification methods, the size of dataset targeted with this thesis means that memory usage and processing time are possibly more important. It is easy to create a highly accurate simplification given enough time and power, but for massive TINs there has to be a trade-off between how accurate the result is to ensure the triangulation is still created within a reasonable amount of time. Further elaboration on this trade-off is included in the discussion; Section 5.3.

3.3.1 Accuracy

It is necessary to determine the accuracy of the resulting simplified triangulation to assess whether the result produced by the simplification method is still within the desired specified vertex error threshold. This measurement is used to determine whether a simplification method is performing as expected. All methods should perform close to the specified vertex error threshold, else they should not be used.

To determine the accuracy of a simplified TIN, the result is compared to the input dataset on a point-by-point basis. Initially, the simplified TIN is loaded, after which for each point in the original dataset the z-error of the point within the simplified TIN is determined by comparing the linearly interpolated z-value within the TIN to the actual z-value. The difference between the interpolated z-value and the actual z-value is stored as the error, which results in an output containing all the original points from the dataset with their respective z-error value.

All interpolation done in this research is done with *linear* interpolation. Linear interpolation works by finding which triangle contains a vertex and subsequently using the elevation values of the corner



(a) How elevation of a point is estimated using the triangulation.

(b) Calculation used to determine the elevation of a point within a triangle.

Figure 3.14: How linear interpolation of a vertex in a TIN works. Figures from Ledoux et al. [2021a].

vertices of that triangle to estimate the value of the requested point. Figure 3.14 provides an example of how the distances between each corner point and the requested point are used to determine the elevation value (x) of the request point.

Further relevant statistics that can be created using this information are the RMSE and the maximum error. RMSE is used as a statistic to determine whether the simplified TIN is close to the user-specified threshold and thus works as a simplification method. The maximum error is used to show the spread of the error values. Using this can show whether some methods may produce a higher RMSE but a lower maximum error or a lower RMSE with a higher maximum error. In most use-cases a low RMSE with a high maximum error is acceptable, but for some it may be necessary to achieve a low RMSE as well as a low maximum error.

3.3.2 Computation Time

Computation time is a relevant measurement because the simplification methods need to be able to process large datasets in a reasonable amount of time. Whether it is reasonable in this case is measured as a situation in which the computation time does not exceed 4 hours per AHN3 tile. Despite this target, processing the entire AHN3 dataset would still take around 170 days to complete, which is an important observation that is further discussed in Section 5.3 (discussion). This threshold is currently chosen to ensure that the methodology chosen is applicable to be used on the target dataset of up to 8 tiles without exceeding the time limitation of this research.

Computation time is measured by comparing the Unix time at the start and end of the entire streaming pipeline. This means that comparison includes the initialization of the finalizer and completion of the triangulator. These programs will remain unchanged between the various simplification methods to ensure a fair measurement between all methods.

3.3.3 Throughput

The throughput is a measurement of the average number of points per minute a simplification method can process. This is based on the number of points that can be processed by the simplification method from the standard input. This measurement is taken just prior to the simplification method starting its processing. This works as a measurement, because while the simplification method is busy it won't accept new input from the pipeline, thus providing a fair measurement of how many vertices can be read by each method.

Throughput is used in combination with the computation time to determine how efficient a simplification method is in simplifying the input dataset. This is because a low computation time with a high throughput could mean that the simplification method is not removing as many points as desired. An opposite situation is also possible in which a high computation time combined with a low throughput means that the method is very efficient in simplifying the dataset, but it takes a long time to do so. Therefore the relation between these two measurements can, to a certain extent, be defined as what the *efficiency* of a simplification method is.

3.3.4 Memory Usage

Memory usage is logged to ensure that the memory usage is still within reasonable bounds when applying simplification within the streaming pipeline. The advantage of a streaming pipeline is that it uses very little memory, therefore it would be counter intuitive to have a simplification method that increases the memory usage by too much. In this case 'too much' is measured as 'no longer fitting in the main memory of a modern desktop computer'. With the increase in memory module size since the introduction of the streaming pipeline (2006), this upper limit can be adjusted to a maximum of 8GB of memory usage for the entire streaming pipeline, including a simplification method. This ensures that the aim of being able to process large datasets on regular computers is achievable, as 8GB of RAM is currently commonly found on many workplace PC's.

3.4 Dealing with Artefacts on Quadtree Borders

All of the simplification modules that are implemented at position B use the quadtree cells as a way to clearly simplify highly spatially coherent sections of the dataset. This is an advantage because it allows for each cell to be processed in parallel, further discussed in Section 4.2.1, thus reducing the time it takes for the calculation to complete. However, a disadvantage is that simplifying within these quadtree cells can result in (visual) artefacts. An artefact can be a clear cell boundary along each cell, which has no effect on the accuracy but is visually obtrusive. Another artefact that can occur is that a larger error is measured along boundaries of quadtree cells due to difficulties in stitching the results of the cells together.

To combat the latter artefact, the streaming pipeline waits with finalizing the vertices that are on the border of a quadtree cell until the parent cell is ready to be released. This means that the vertices that connect two neighboring cells are only released when both cells are finalized, ensuring that there are always accurate vertices to connect quadtree cells.

The visual artefact will occur mainly because a simplified triangulation highlights the four corners of a quadtree cell, which is then also seen in the final result. This behavior occurs because the vertices nearest to the corners are essential in determining whether a triangulation is accurate, as these are used to interpolate points within the cell. These artefacts may be clearly visible while the triangulation is still being simplified, as each cell that is released at that moment in time will not directly be connected to a subsequent cell. However, once a cell is connected to another cell the vertices along the border are finalized, which ensures that the sharp edges of the quadtree cell are connected to more vertices. By doing so these edges also nearly completely disappear, thus resulting in a simplified triangulation with hardly any (visual) artefacts.

It is important to take into account that even if there are clear boundaries along the borders of a quadtree cell, the triangulation is still accurate locally as these borders represent the triangulation within. This means that despite the result looking artificial, the validity of the result is no less true. In Figure 3.15 an example is seen of how much of the quadtree cell borders remain for FCFS simplification on a large sample set.

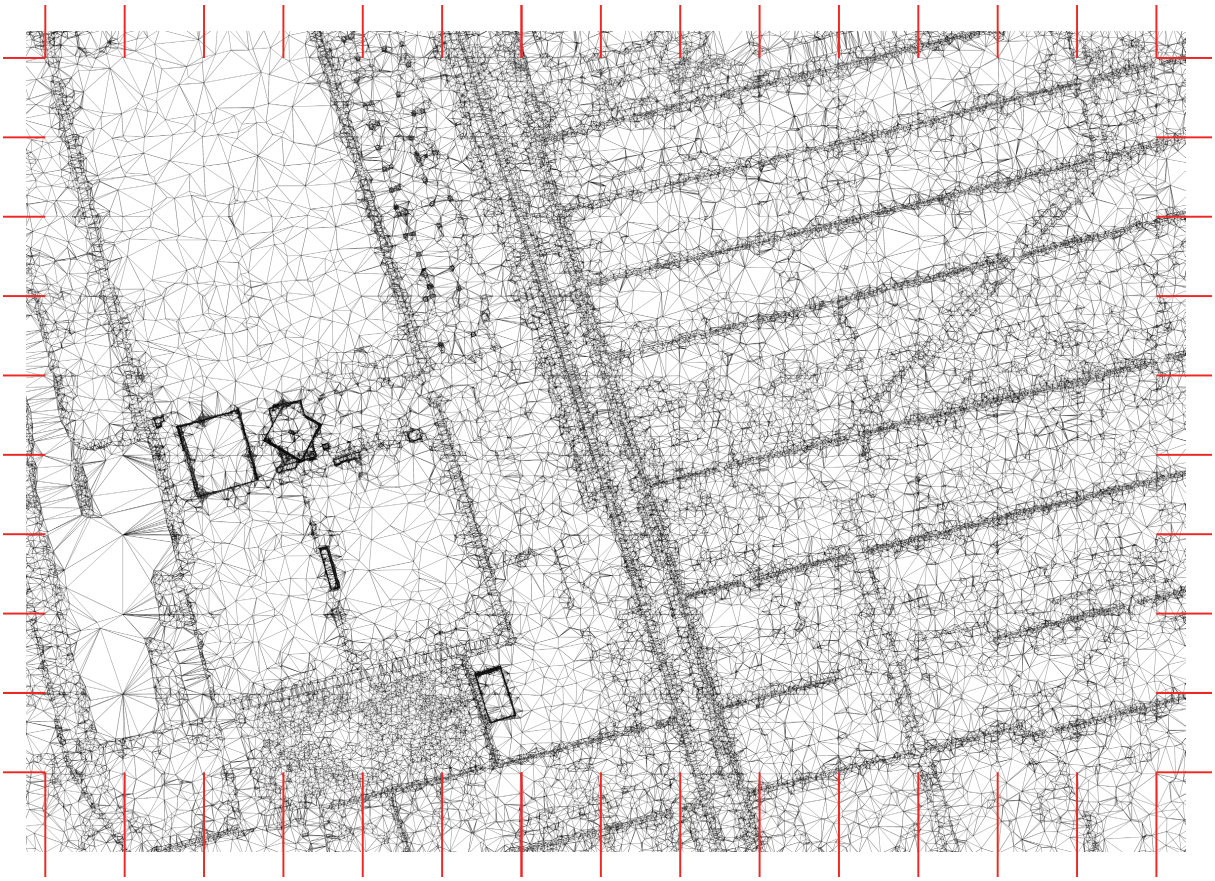


Figure 3.15: The level of visual artefacts that can occur for simplification methods implemented at position B. Red lines denote cell boundaries and are shown to assist in detecting visual artefacts.

4 Implementation and Results

This chapter discusses the specific details related to how the methodology is implemented in practice, including examples. These implementations can be found here in a GitHub repository, freely accessible to anyone interested. Furthermore, this chapter shows all the obtained results and which methods performed well enough to be used on a large scale. In Section 4.1, it is explained how the evaluation criteria discussed in Section 3.3 are implemented and how they can be visualized. This is followed by in-depth information relating to how the implementations have been written in code in Section 4.2. Here, information is provided on how the multi-core capability of most CPU's is leveraged to improve the processing speed of simplification modules. In Section 4.3 the datasets which are used to validate the simplification methods are introduced before the results for each simplification method are discussed in the final section of this chapter; Section 4.4. This includes an overview of how the created simplification methods compare to the other relevant researches discussed in Chapter 2.

4.1 Evaluation Tools

In this section the tools used to evaluate the results of this research are explained. These tools are necessary to retrieve the statistics used to evaluate the functioning of the simplification modules. These statistics are retrieved using a number of different methods; accuracy is calculated after processing using another script and memory is logged while each simplification algorithm is running.

4.1.1 Time measurement

Measuring how long a simplification module takes to process a dataset may seem straightforward, but it is necessary to define which time is measured for this research as ambiguity is possible. Time needed by a process to complete can be measured by using the *time* command on Linux, which will display an overview of the time taken, as seen in Figure 4.1. This shows three different times; real, user, and sys(tem) time. The real time is the amount of time that has passed on a clock. Combining user and system time results in the amount of CPU time that is spent running the program. CPU time is measured on a per-core basis which means that if a program uses multiple cores the user+sys time can exceed the real time.

```
1    real    142m50.959s
2    user    217m30.334s
3    sys     35m45.572s
```

Figure 4.1: Example of the output from the *time* module in Linux showing real, user, and sys(tem) time.

For this research the choice is made to use the real time as a measurement for the time taken to triangulate and simplify a set of points. This decision is made because some of the methods are less efficient, but can be run in parallel (more about this in Section 4.2.1). These processes use a lot of CPU time, but can run a lot faster in real time. Furthermore, the real time is used because in the end it is relevant to know how long the wait is for a result.

It should be noted that the time taken by the finalizer and triangulator is also taken into account within the time measurement. However, because these run much faster than the simplification methods the

total time a simplification module is active is nearly identical to the real time measured including the finalizer and triangulator. This has been tested by comparing the active time reported by the simplification module with the time reported from the *time* command.

4.1.2 Accuracy

To measure the accuracy of the simplified TINs created by the simplification modules, 1 in 10 points from the input dataset are randomly selected and measured against the simplified TIN. The measurement is done using linear interpolation of the elevation value in the TIN with Startin, which is a library used to easily create Delaunay triangulations in Python. Error is measured vertically, as seen in Figure 4.2, this means that it does not necessarily take into account the minimum distance of a point to the TIN. Using the minimum distance between a point and the TIN is a good accuracy measurement because it allows for more freedom in determining the vertex error. The reason for not using the minimum distance between a point and the TIN to measure accuracy is because not all parts of the TIN are loaded into memory at all times. This could result in the point being closest to a triangle in the TIN already having been written to disk, thus returning an unknown value or a poor estimate. Furthermore, using the minimum distance between a point and the TIN increases the computational complexity of the simplification methods, resulting in a slower simplification.

Figure 4.2: Figure showing that error is measured vertically and how a TIN subsequently approximates the real surface. Adapted from Ledoux et al. [2021b].

From these error measurements, a GeoJSON is created which contains every tested point with an attribute that indicates the accuracy. This method is used because it allows the accuracy to be visualized in QGIS by plotting the points and defining their error property as a heatmap. An example of what this looks like is seen in Figure 4.3.

This heatmap is generated by QGIS and uses the location of a point to draw a circle of a user-specified radius around that point. The color of the circle is defined based on the accuracy value of the point, ranging from transparent for 0 - 0.2m error to dark red for any errors above 3m. This range is defined as such because errors within the threshold are not relevant for the assessment and any error of 3m or more is significant and should be clearly highlighted.

This information can also be used to extract information about the RMSE and the deviation of the errors, as the number of occurrences combined with the error is available in the GeoJSON dataset. For this, a Python script is made which takes the GeoJSON of the errors and processing this into a plot with the relevant data indicated. An example of such a plot is seen in Figure 4.4.

For the datasets that are an entire AHN3 tile or more, the accuracy is measured by taking a subset from the entire tile or set of tiles. This is done because the size of the TINs that result from these datasets are too large to retrieve this accuracy measurement for an entire tile. Within each tile the same segment is chosen but the vertices that are tested from the original dataset are randomized.

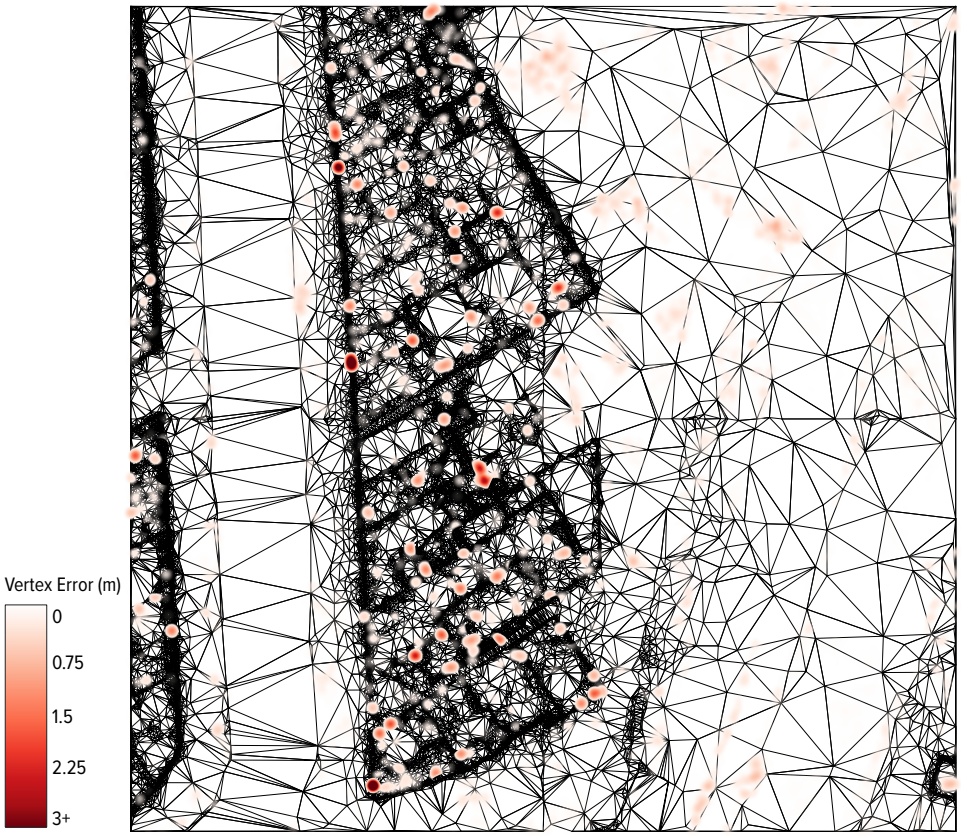


Figure 4.3: Example showing how QGIS is used to plot the error of each tested vertex on top of the simplified TIN for the smallest test set.

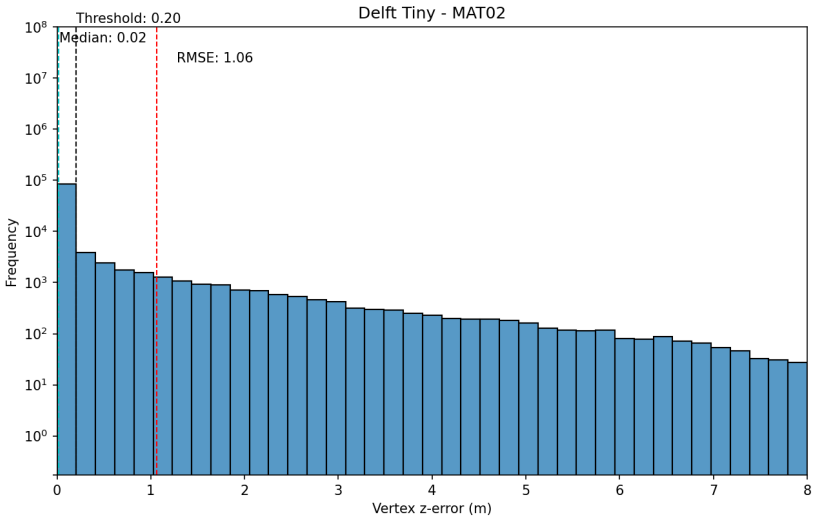


Figure 4.4: Example showing what plotting the histogram looks like. Vertical lines added to show relevant values: threshold, median, and RMSE.

4.1.3 Memory Analysis

Memory analysis is necessary to ensure that the principle of the streaming geometries paradigm is not broken by adding simplification to the pipeline. As a reminder, the paradigm exists to ensure that large datasets, that do not fit in memory, can still be processed. Therefore, if simplification uses more memory than is available on the system, the method is not suitable for use within the pipeline. It should also be noted the memory usage stated in the results is the memory usage of just the simplification modules, excluding the memory usage of the finalizer and triangulator. The memory usages for the finalizer and triangulator have been identified in Section 2.3.2 and Section 2.3.3 respectively and can be taken into account if necessary.

Memory analysis is done by making each process that is running in the simplification module output its own memory usage to a common queue. Another process then writes these messages from the common queue to a comma separated values (CSV) file for later plotting. This data can then be plotted for each process against the timestamp to show what the memory usage per process and total memory usage of the simplification module is. An example of this is seen in Figure 4.5. Maximum memory usage is identified by taking the largest sum of memory usages found at a given moment in time. Memory is logged every second, thus the second with the highest cumulative memory usage is used as maximum memory usage.

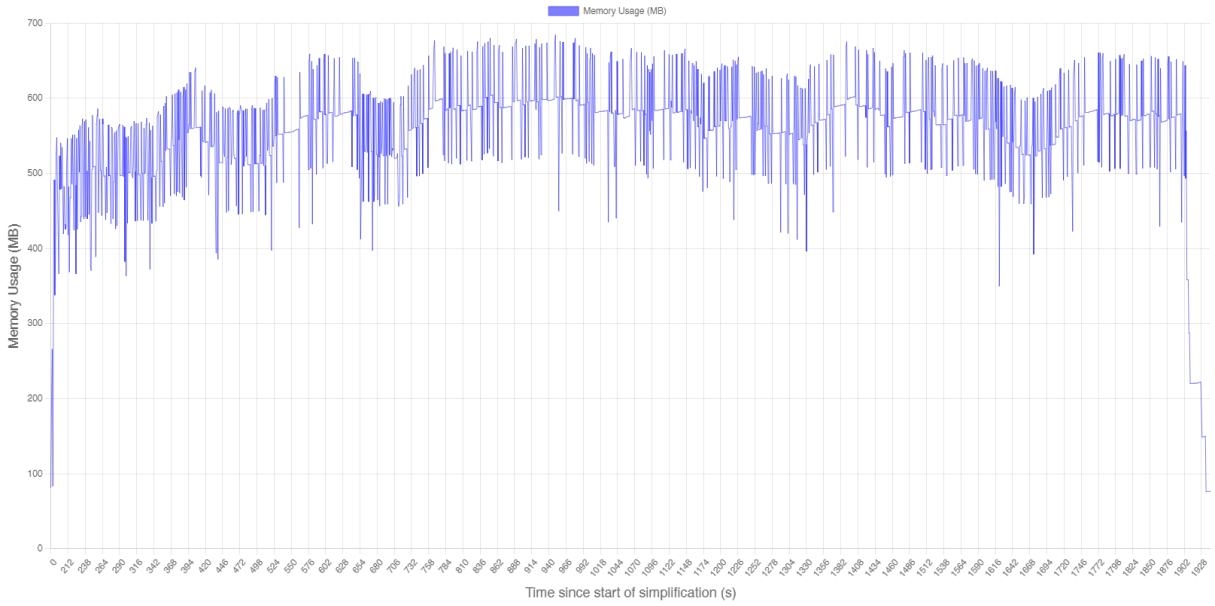


Figure 4.5: Example showing the memory usage plot of a simplification module may look like.

The plot is created by inserting the memory usage CSV that is created by each simplification method into a structured query language (SQL) database and plotting this using JavaScript in a browser. This method is chosen because the dataset is quite large and storing this in a SQL database allows for easy manipulation of the data. This means it is much easier to sum or exclude data compared to using Excel or writing an analysis script in Python.

4.1.4 Streaming Geometries Visualizer

As part of this thesis, a simple visualizer is created to show which geometries are being created by the streaming pipeline in real-time. This tool is written in Python using the PyGame library which is capable of rendering 2D graphics to a window. The visualizer, dubbed *sstvis*, can be plugged into the streaming pipeline at position C, but with adjustments can also be used in position A or B. In Figure 4.6 a sample

image can be seen of geometries being streamed into *sstvis* in real-time. Statistics showing how long the visualizer has been live as well as how many points and triangles are in the triangulation are also updated as data is received.

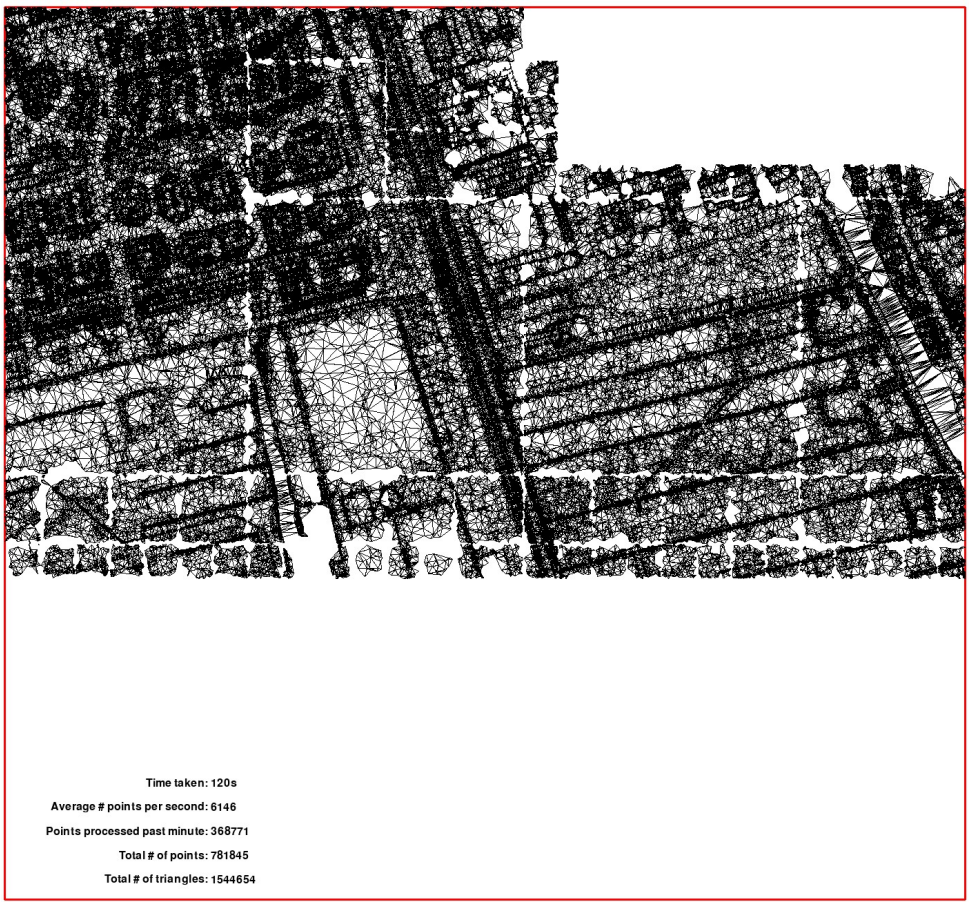


Figure 4.6: The capability of *sstvis* to render streaming geometries in real-time. The red line denotes the bounding box of the dataset which will be entirely filled by the triangulation by the time the pipeline is finished. The quadtree structure can clearly be seen while rendering.

4.2 Engineering Decisions

This section discusses some of the specifics around the choice of programming language as well as choices made to increase the speed at which the simplification methods are able to process data. For all simplification methods Python is used as main language as it allows for rapid prototyping without dealing with typing variables and compilers. Python has some limitations in speed as it is an interpreted programming language, as opposed to a compiled language such as C++ or Rust. This means that it takes more time to evaluate some functions or types of variables. The reason Python is still used as the main language for the simplification methods is for two reasons: to show how easily a simplification module can be written for this streaming geometries pipeline and because most of the functionality used is actually C or Rust in the underlying code.

Many libraries that are written in Python are actually written in other languages and include a wrapper to allow Python to communicate with the internal functions. This is the case for the max heap which is used in some simplification modules, where C is the underlying language. This is also the case for Startin, a library used for creating and interpolating Delaunay triangulations, where Rust is the

underlying language. These implementations will always be slower than writing the entire simplification method in either C or Rust, but Python is a useful language to bring all these separate elements together.

Combining libraries written in other programming languages in Python together with the use of parallelization, discussed in Section 4.2.1, shows how accessible the processing of massive TINs can become using the streaming geometries paradigm.

Each of the simplification methods relies on reading data from the standard input, processing that data, and subsequently releasing it to the standard output. Depending on the position of the simplification method in the pipeline the internal processing of the input data differs, as well as the format in which the result is released to the standard output. The formats within which data is read and written is as discussed in Section 3.2.3.

4.2.1 Parallelization

Figure 4.7: Possible placement positions for a simplification module within the *sst* pipeline.

One of the approaches used to increase the processing speed of the simplification algorithms at position B in the pipeline (see Figure 4.7 for a reminder), is to finalize each cell in a separate process. This can be done because the finalizer releases all vertices on a cell-by-cell basis. This means that depending on the number of cores available on the CPU a large number of simplifications can be run in parallel. An example of how this works is seen in Figure 4.8. It should be noted that only one process is allowed to release its result to the output stream at a time to ensure that written lines do not clash with one another. If processes are to write to the standard output at the exact same time this results in corrupt data and the triangulator will crash. Essentially, parallelization is related to the divide-and-conquer technique introduced by Wu et al. [2011] which is discussed in Section 2.1.

The user is able to define how many cores should be used, though the recommended limit is just below the total number of cores. This is because a system will always want some overhead for other processes, allowing each core to work at full capacity. What is also important to know, is that when all processing slots are in use, the simplification module is unable to process new data. The effect is that the entire pipeline is halted until a process is completed and a processing slot is opened up. An advantage is that data is not buffered into memory while waiting for a processing slot to free up, thus maintaining the low memory characteristic of the streaming pipeline. Instead the first process in the pipeline is literally stalled until it is able to release data further.

Using parallelization for simplification modules in position A or C is more challenging because modules rely on all available points, as opposed to merely the points present within a quadtree cell. This is because points are simplified after every n insertions and not in spatially coherent cells. Because not all points are inserted as connected components each time the internal simplification function is run, simplification heavily depends on reusing points from previous iterations as reference data. Therefore it is not possible to apply parallelization for the modules implemented in these positions and thus they are significantly slower in comparison to the simplification modules in position B.

4.3 Real-World Datasets

The goal for this research is to triangulate and simplify up to 8 AHN3 tiles using the streaming geometries paradigm. However, 8 AHN3 tiles is 250km^2 and thus approximately 4 billion points, or a little over 18 GB of compressed LIDAR data (LAZ). Therefore, running this dataset several times for testing

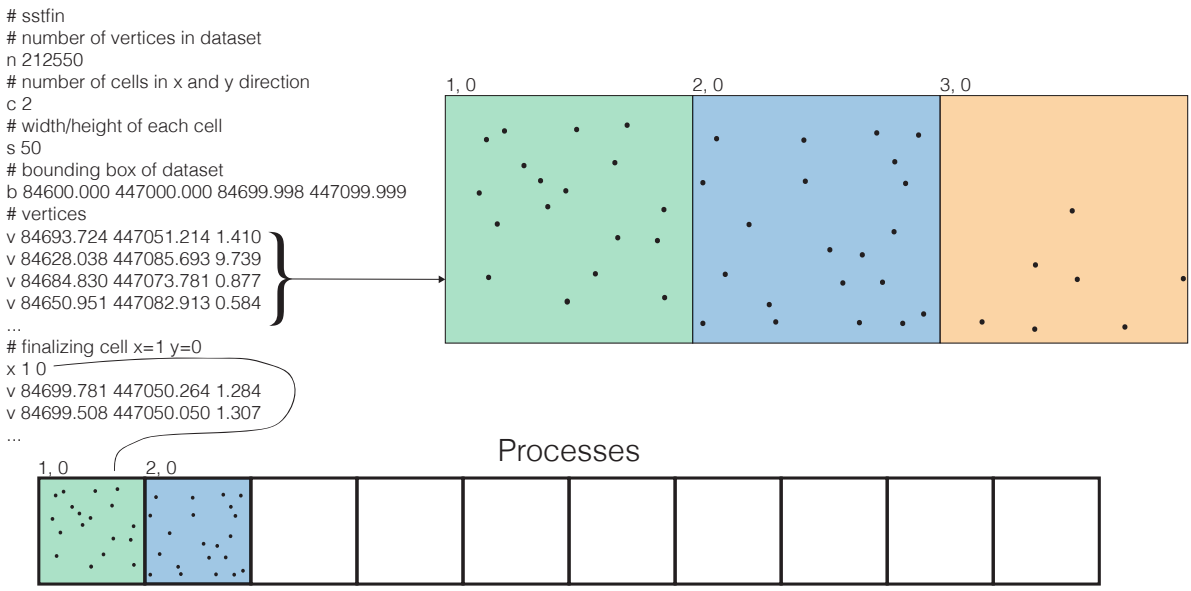


Figure 4.8: Vertex data being stored into a cell which is subsequently simplified in a separate process when a finalization tag is received. In the meantime the filling of a new cell can continue and finalization can be done until all processor slots are filled.

is hardly practical, so a number of test sets have been chosen that are representative areas which can be processed quicker to show possible issues with the simplification methods and to show how these methods scale with the size of the dataset. The following sections elaborate on the different test sets that have been chosen. The following three types of tests have been deemed necessary to validate the workings of the simplification techniques:

1. Frequent tests
2. Less-Frequent tests
3. One-time tests

4.3.1 Frequent tests (Small-Scale)

The datasets used for frequent tests are relatively small datasets. The first dataset is a subset of the 37EN1 tile located at Delft and contains 212,550 points. This dataset can be processed in about two seconds when using sst on a cell size of 50 without adding a simplification module. The second dataset used for frequent tests contains approximately 28 million points and can be triangulated using sst at a cell size of 50 in about five minutes. This dataset is also a subset of the 37EN1 tile at Delft. Both subsets are seen in the top left corner of Figure 4.9, and are marked in red in the larger overviews. The smallest of the two datasets is 0.01km^2 and is further referred to as *Delft-Tiny*. The larger of the two is 1.4km^2 and is further referred to as *Delft-Small*.

These specific clipped sections are chosen because both contain a mix of water, buildings, and height differences in the terrain. Their size is chosen to contain a relevant section of data while still being small enough to be processed on a frequent basis. These datasets are meant to be run multiple times a day after each (small) iteration to the simplification methods has been made. The aim is to be able to validate the quality of the result of the simplification methods.

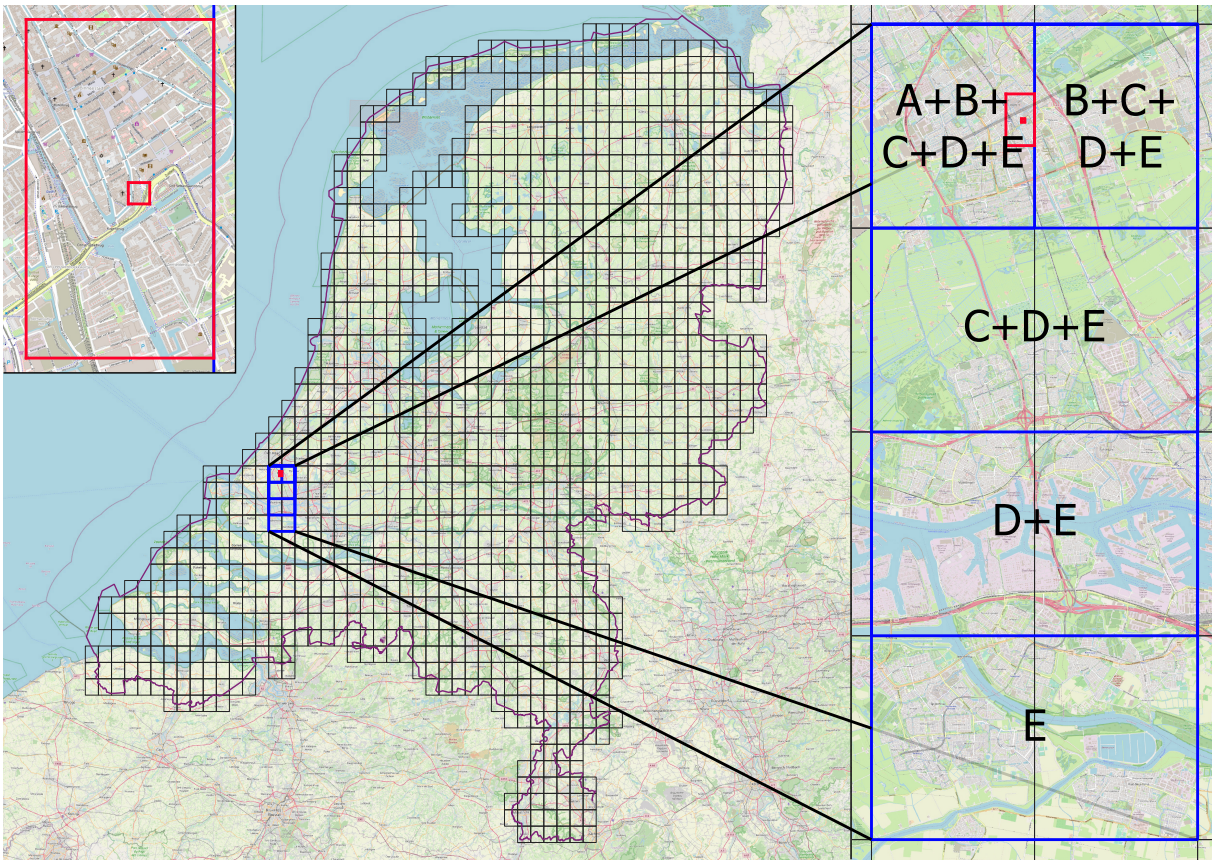


Figure 4.9: Overview of all the datasets used.

4.3.2 Less-Frequent tests (Large-scale)

The large-scale tests are made to be be run on a less-frequent basis to validate the methodology for its processing speed when simplifying large amounts of data. This dataset consists of either a single AHN3 tile, 37EN1 at Delft, but can be extended with 1-3 more tiles to cover 125km^2 by adding 37EN2, 37EZ1, and/or 37EZ2. The total number of points in these four tiles is just under two billion. These four tiles represent half of the desired target area and therefore are used to validate the functioning of simplification prior to running an even more extensive test.

The dataset consisting of a single tile is referred to as the *Single Tile* dataset in the results, or in Figure 4.9 as A. The dataset with two tiles consists of 37EN1 and 37EN2 and is referred to as the *Two Tiles* dataset in the results, or dataset B in Figure 4.9. Lastly, the dataset with four tiles consists of 37EN1, 37EN2, 37EZ1, and 37EZ2 and is referred to as the *Four Tiles* dataset in the results, or dataset C in Figure 4.9.

4.3.3 One-Time tests (Full-scale)

The full-scale dataset (8 tiles) covers 250km^2 and contains approximately 4 billion points. For this reason this dataset is only processed once to show the capabilities of the streaming geometries pipeline combined with simplification. This dataset is chosen because it provides a challenging combination of AHN3 tiles that are mostly water, tiles that are separated by the river Maas, and tiles containing large cities with many buildings of various sizes. Processing this dataset is a combination of validating whether the streaming simplification method functions well for large-scale triangulations but moreover

it tests whether the sst streaming geometries pipeline is capable of dealing with very large Delaunay triangulations. This application has yet to be validated on such a scale and therefore this test set is perfect to combine the validation of both simplification and sst.

This dataset is referred to as *Eight Tiles* in the results, or as E in Figure 4.9. Here it can clearly be seen how the segment stretches across a body of water and is quite a narrow, tall, dataset. The *Six Tiles* dataset is also a one-time run and is marked as D in Figure 4.9.

4.4 Results

In the following sections the results per test set are discussed. First of all, the results for the small-scale datasets are shown and analyzed. The small-scale datasets are the only datasets that can be visualized as a whole because they are small enough to be rendered. For all the larger datasets a section is clipped from the large TIN for analysis. For each of the larger datasets the same analyses are done as for the small-scale datasets. Here, the focus will mainly lie on differences between the small-scale results compared to the larger-scale results.

All tests are run using an Intel Core i7-8700K 6-core, 12-thread, processor running at 4.8GHz with 32GB of 3000MHz RAM. A 1TB solid state drive (SSD) with 3400MB/s reads and 2300MB/s writes is used as file storage. With simplification modules that support multiprocessing the maximum number of cores used is 8. For each test the threshold accuracy for vertex z-error is 0.2m. The amount of randomized thinning is chosen to achieve a number of vertices similar to that of the other simplification modules which allows for fair comparison between the resulting values. Furthermore, the finalizer uses a cell size of 50m and uses only points that are classified as building, ground, or water, effectively ruling out vegetation and unclassified points.

Table 4.1: Statistics on the input datasets. All sets contain ground, water, and building points with exception of the last.

	area	number of points	average density
Delft-Tiny	0.01km^2	212,550	21.3 points/m^2
Delft-Small	1.40km^2	20,310,070	14.5 points/m^2
One Tile	31.25km^2	343,044,606	11.0 points/m^2
Two Tiles	62.50km^2	642,167,999	10.3 points/m^2
Four Tiles	125.00km^2	1,354,295,996	10.0 points/m^2
Six Tiles	197.50km^2	1,902,652,509	9.6 points/m^2
Eight Tiles	250.00km^2	2,424,250,813	9.7 points/m^2
Dukai Comparison (only ground)	13.89km^2	118,552,841	8.5 points/m^2

4.4.1 Small-Scale

This section shows the results obtained for all the possible simplification methods when applied to the Delft-Small dataset of 212,550 points. Table 4.2 includes the number of output vertices and triangles, indicating the level of simplification that has occurred as an effect of these simplification methods. Furthermore, these measurements are used to determine whether some methods are performing so bad that they can be excluded from further tests. The decision to remove methods from further testing is based upon the expectations set in Section 3.3.

Only one decimation method is present in this test set as the time taken at a recalculation interval of once every 100 points is already too large, thus reducing this interval will not improve this metric. Besides

Table 4.2: Delft-Tiny: Results of the simplification methods.

	time taken	vertices	triangles	RMSE	max error (m)	max memory (MB)
No Simplification	2s	212,550	425,046	-	-	-
Rand5	1s	35,318	70,600	0.94	10.3	12
Decim100	27m25s	53,345	104,999	0.23	8.4	675
Greedy025	1m25s	37,651	75,270	0.07	5.2	357
Greedy1	44s	39,994	39,957	0.09	5.3	362
Greedy5	23s	44,570	89,109	0.06	1.4	358
FCFS	1s	34,704	69,395	0.13	6.2	337
FCFS + Decim10	8m30s	26,419	52,825	0.31	7.8	400
MAT02	15s	14,551	29,057	1.06	11.0	142

this, it is possible to increase the interval to improve the time taken, but this would result in missing the desired RMSE of around 0.2m.

The histogram displaying the errors for each of the results can be seen in Figure 4.10. What is apparent is how the greedy refinement methods show the least spread of vertex error followed by decimation. FCFS follows closely and performs surprisingly well for how fast the method is. Contrary to that, FCFS+Decim10 performs surprisingly bad considering it is made to improve upon the result of FCFS alone. This is possibly due to the 1-in-10 recalculation interval used, which results in relevant points being removed in one of the batches of 10.

Based on the results obtained from the Delft-Tiny dataset (Table 4.2), the most promising methods are greedy refinement and FCFS refinement. The RMSE for MAT simplification with an ϵ of 0.2 is too far above the desired threshold to be considered as an accurate enough method. This is supported by the statistic that random thinning provides similar results to the MAT method. Notably, the level of simplification for the MAT method appears to be too much when comparing the number of remaining vertices and triangles. Therefore, attempts are also made for MAT simplification using lower ϵ values of 0.1, 0.05, and 0.001. Unfortunately these lower error thresholds show no significant improvement in error metrics nor do they increase the number of vertices and triangles by a large amount. This may suggest that the ϵ input argument of this method is not working as intended.

Furthermore, the time taken by the decimation method is also too far above the other methods and is therefore not considered as a viable method for further analysis. The time taken would greatly exceed the desired half a day processing time per AHN3 tile. Despite this, these methods are still tested for the Delft-Small dataset to determine if the methods may perform differently at a larger scale. The results of this are seen in Table 4.3.

Table 4.3: Delft-Small: Results of the simplification methods.

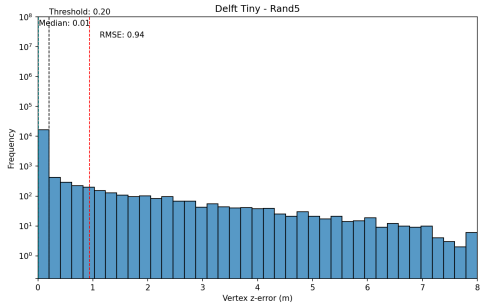
	time taken	vertices	triangles	RMSE	max error (m)	max memory (MB)
No Simplification	3m30s	20,319,438	40,638,770	-	-	-
Rand3	1m37s	5,079,401	10,158,749	1.38	30.0	12
Decim100	>18h	-	-	-	-	-
Greedy025	29m26s	5,021,178	10,042,286	0.12	12.3	684
Greedy1	17m15s	5,333,951	10,667,831	0.10	9.0	630
Greedy5	9m12s	5,970,399	11,940,721	0.11	10.9	773
FCFS	4m4s	4,916,929	9,833,779	0.20	15.0	491
FCFS + Decim10	8h4m	3,614,934	7,229,797	0.46	24.0	385
MAT02	14m5s	2,236,205	4,472,145	1.53	55.7	345
Full Refinement	17h53m	4,210,394	8,420,723	0.16	20.0	673

As can be seen in Table 4.3 the Decim100 method has no results. This is because the module took over 18 hours to process before it was decided to break off the run. Therefore, this method is definitively removed from further analysis as the runtime is far too large to realistically continue using it for larger datasets.

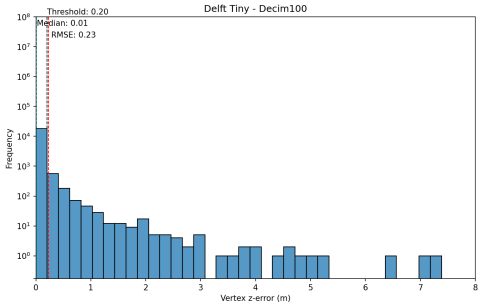
In Table 4.3, a result for *full refinement* can also be seen, which is added as a comparison method for the Greedy methodologies. The full refinement method relies on a static recalculation interval that does not increase while calculating and is based on recalculating all vertex errors on every insertion. As can be seen, the time taken with this methodology is much longer when compared to the Greedy methods, resulting in an execution time of almost eight-teen hours compared to less than 30 minutes at worst. This shows how important the addition is of the increasing recalculation interval that is being used by the Greedy methods as explained in Section 3.2.3.2.

In addition, the RMSE and max error of the result do not differ much from the other methods with an 0.20m RMSE and 20.0m max error. Why this is the case is most likely due to the implementation of the refinement methodology which is geared towards being capable of processing as many points as possible and is less optimized towards accuracy.

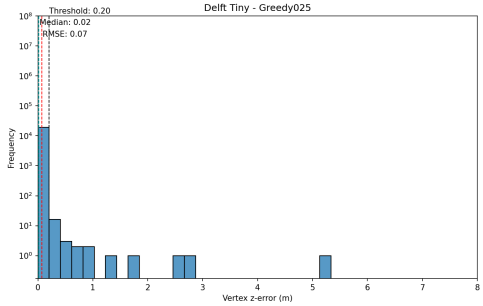
What is apparent from the histograms seen in Figure 4.12 is that it is clearly more difficult to achieve a lower spread of errors with a larger, more diverse, dataset. This makes sense, as when a dataset becomes more complex, it is more difficult for the simplification methods to create accurate representations using only local information. However, it is good to see that this expectation is reflected in the results. The greedy and FCFS modules are, once again, the best performing ones, being able to keep the lowest frequency of error occurrences to 0. All the other methods have a lowest frequency of at least 100 or more, thus clearly struggling with the larger dataset.



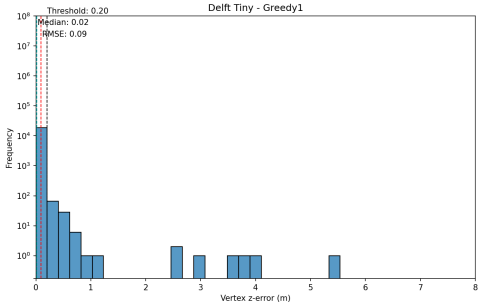
(a) Rand5



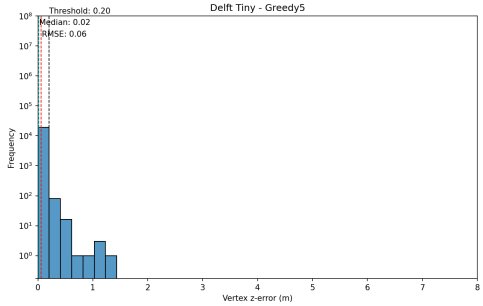
(b) Decim100



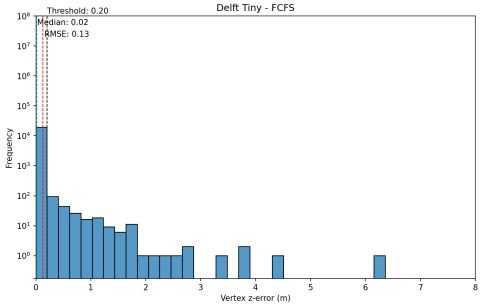
(c) Greedy025



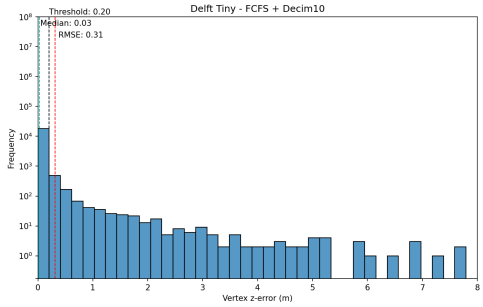
(d) Greedy1



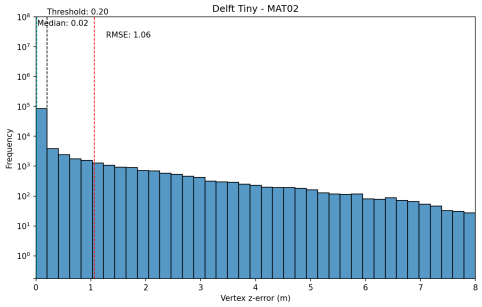
(e) Greedy5



(f) FCFS

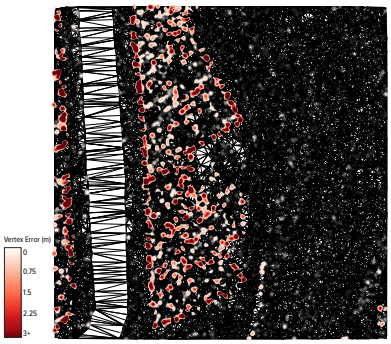


(g) FCFS+Decim10

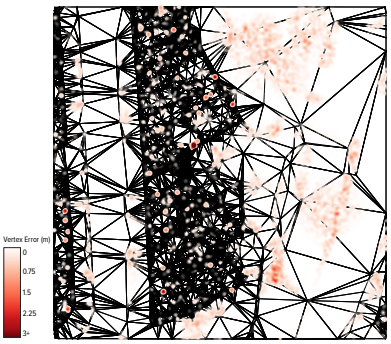


(h) MAT02

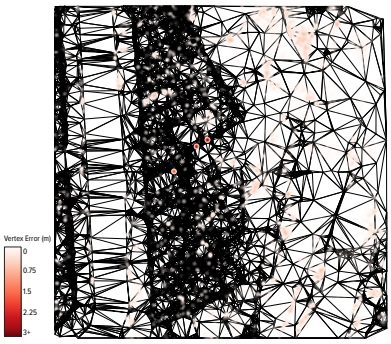
Figure 4.10: Delft-Tiny: Error histograms showing the distribution of error for each method.



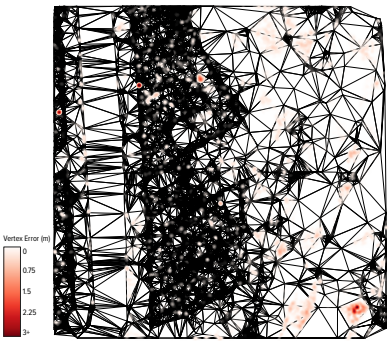
(a) Rand5



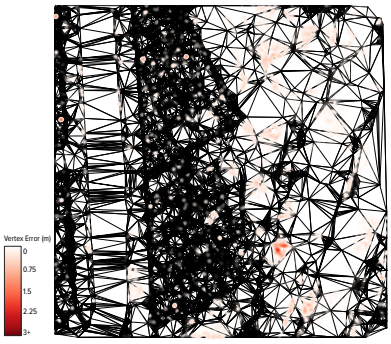
(b) Decim100



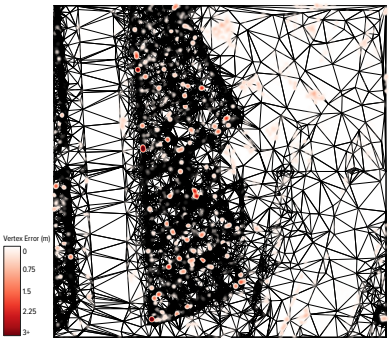
(c) Greedy025



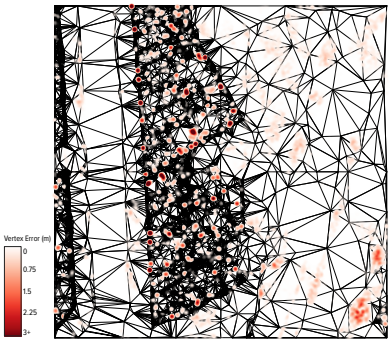
(d) Greedy1



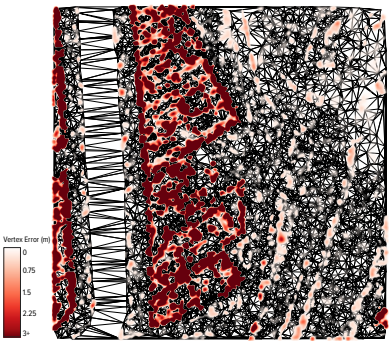
(e) Greedy5



(f) FCFS

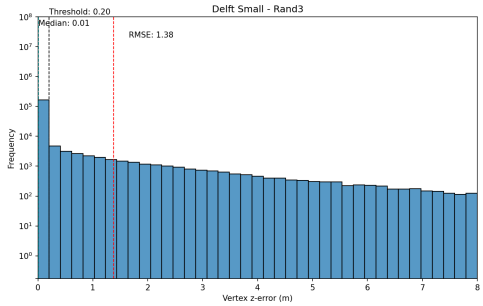


(g) FCFS+Decim10

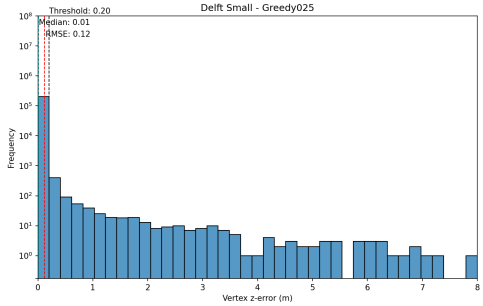


(h) MAT02

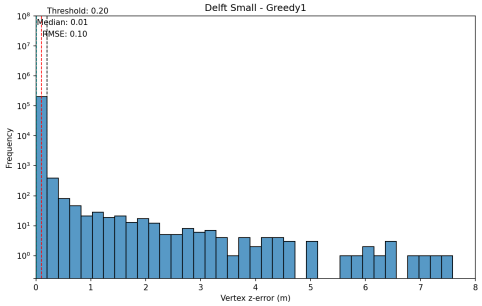
Figure 4.11: Delft-Tiny: Error heatmaps for each method.



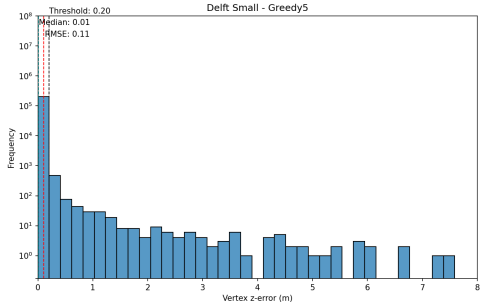
(a) Rand3



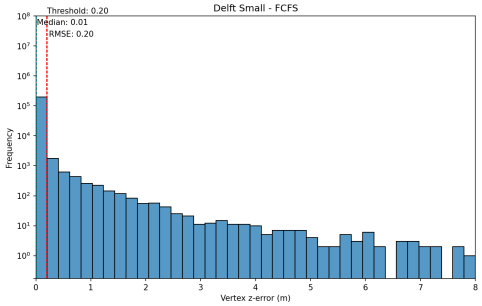
(c) Greedy025



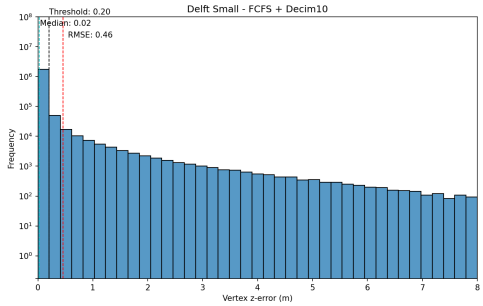
(d) Greedy1



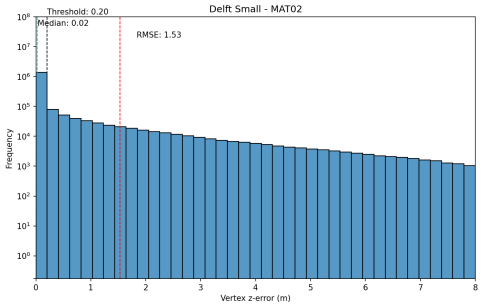
(e) Greedy5



(f) FCFS



(g) FCFS+Decim10



(h) MAT02

Figure 4.12: Delft-Small: Error histograms showing the distribution of error for each method. Decim100 (b) failed to complete in a reasonable time and is therefore excluded from this comparison.

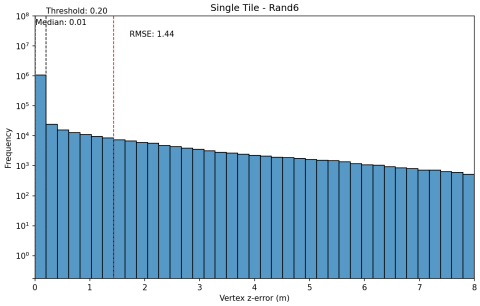
4.4.2 Large-Scale

With the poor performance of the decimation and MAT simplification modules, these are excluded from further analysis with the large scale datasets. This is chosen because their respective computation time and RMSE are much larger than the goals set and the methods show no sign of performing better at a larger scale. Furthermore, because the RMSE and max error of all Greedy methods are so similar, the decision is made to continue the analysis with Greedy1 and Greedy5 due to being faster compared to Greedy001 and Greedy025. In Table 4.4 the results for the Single Tile can be seen for the best performing simplification methods from the small-scale tests.

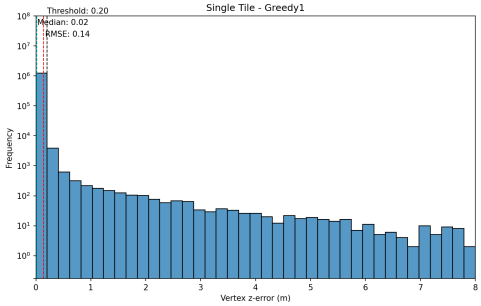
Table 4.4: Single Tile: Results of the simplification methods.

	time taken	vertices	triangles	RMSE	max error (m)	max memory (MB)
No Simplification	1h27m	343,038,929	686,077,602	-	-	-
Rand6	31m10s	48,995,882	97,991,696	1.44	41.8	12
Greedy1	2h34m	45,341,040	90,681,998	0.14	20.7	792
Greedy5	1h49m	53,559,849	107,119,616	0.14	24.1	792
FCFS	1h30m	40,869,309	81,738,535	0.18	20.9	709

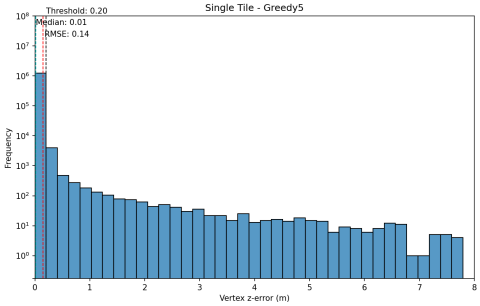
Table 4.4 shows how the various simplification modules perform on a larger dataset. What is positive to see is that the RMSE and max memory usage only increase marginally compared to the results from the Delft-Small test set. There is an average increase in the RMSE of 0.3m, and the maximum memory usage only increases by 200MB in the worst case. This means that the streaming pipeline is working as desired by keeping memory usage stable despite datasets of larger sizes and that the pipeline is capable of producing consistent results in terms of accuracy. However, the maximum error tested is much larger for almost all methods, with a worst-case increase from 10.9m to 24.1m for Greedy5, an increase of 13.2m. Most likely this is related to taller buildings being included in this dataset compared to the Delft-Small dataset.



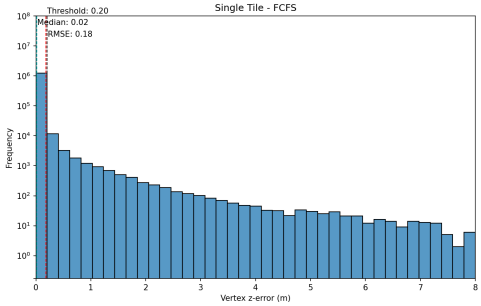
(a) Rand6



(b) Greedy1

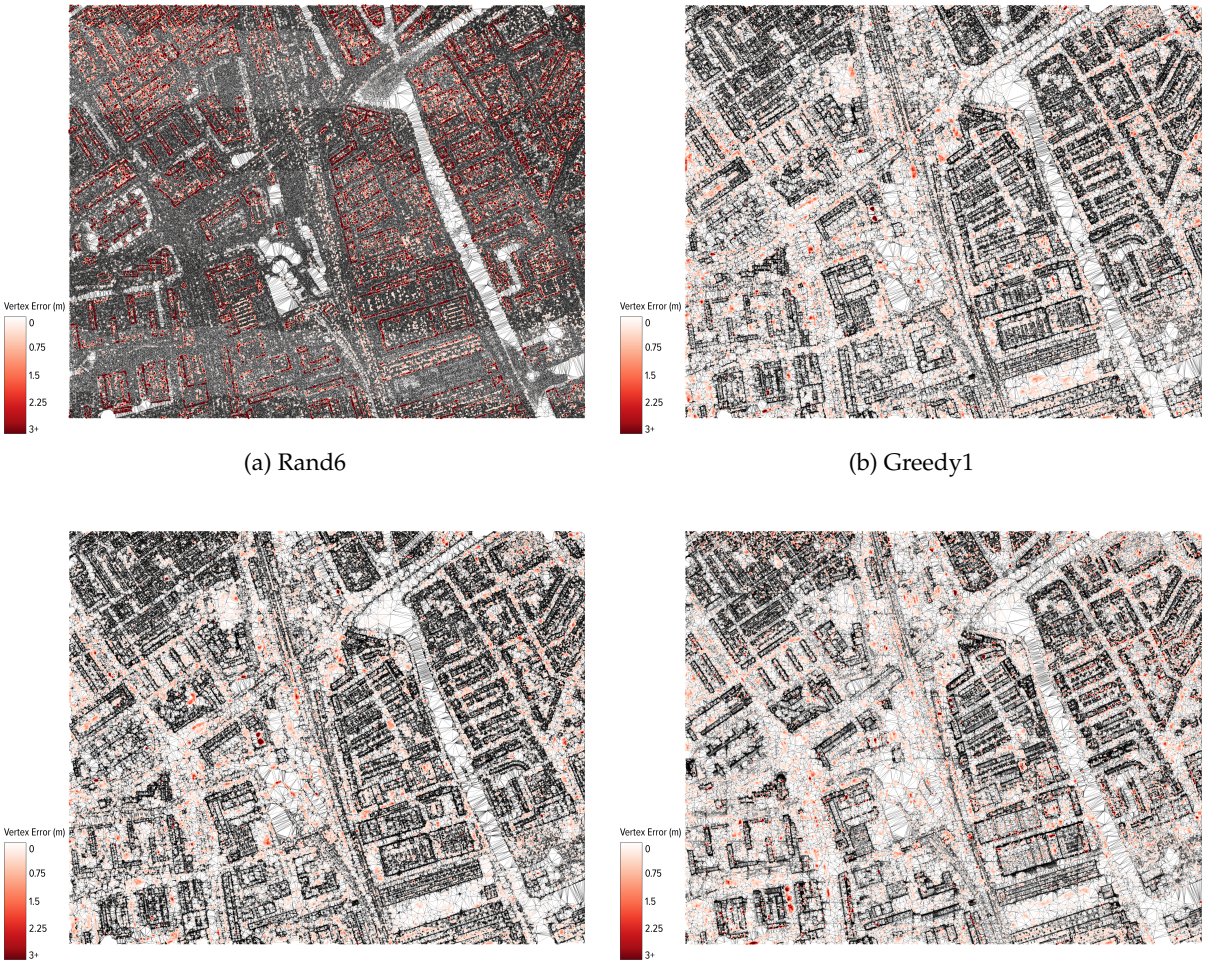


(c) Greedy5



(d) FCFS

Figure 4.13: Single Tile: Error histograms showing the distribution of error for each method.



(a) Rand6

(b) Greedy1

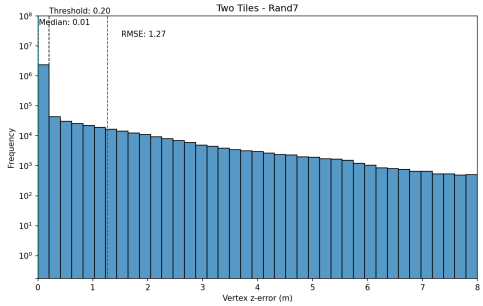
(c) Greedy5

(d) FCFS

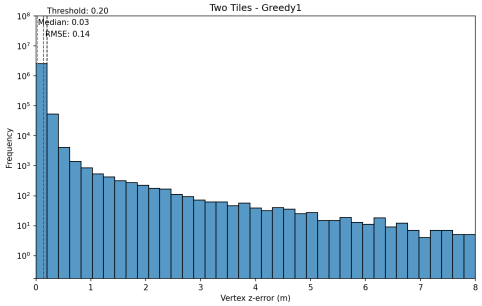
Figure 4.14: Single Tile: Error heatmaps for each method. Zoom in for more detail.

Table 4.5: Two Tiles: Results of the simplification methods.

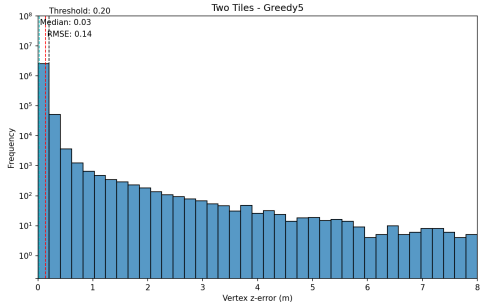
	time taken	vertices	triangles	RMSE	max error (m)	max memory (MB)
No Simplification	2h50m	692,857,134	1,385,713,924	-	-	-
Rand7	37m36s	86,615,876	173,231,679	1.27	50.5	12
Greedy1	5h12m	85,675,406	171,350,670	0.14	31.4	955
Greedy5	3h53m	97,792,691	195,585,235	0.14	33.3	993
FCFS	2h57m	85,292,678	170,585,217	0.16	36.2	698



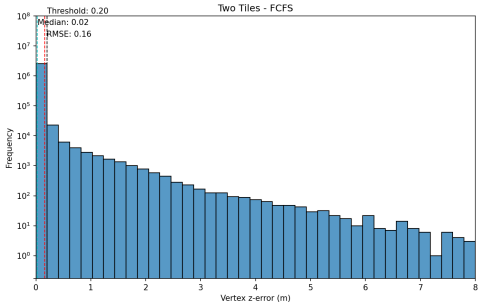
(a) Rand7



(b) Greedy1



(c) Greedy5



(d) FCFS

Figure 4.15: Two Tiles: Error histograms showing the distribution of error for each method.

Table 4.6: Four Tiles: Results of the simplification methods.

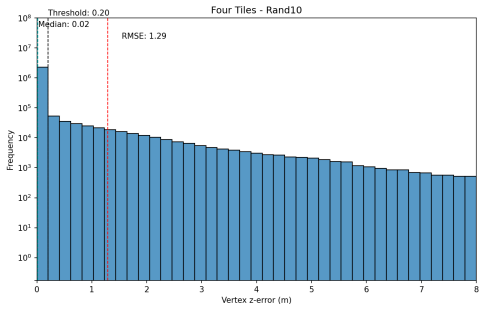
	time taken	vertices	triangles	RMSE	max error (m)	max memory (MB)
No Simplification	5h53m[(1)]	1,354,295,996	-	-	-	-
Rand10	1h15m	127,728,716	255,457,357	1.29	49.3	12
Greedy1	9h39m	132,594,102	265,188,048	0.11	36.2	897
Greedy5	8h32m	159,092,817	318,185,478	0.11	24.9	804
FCFS	5h47m	115,640,355	231,280,566	0.17	42.3	650

[(1)] data piped to `/dev/null` instead of file

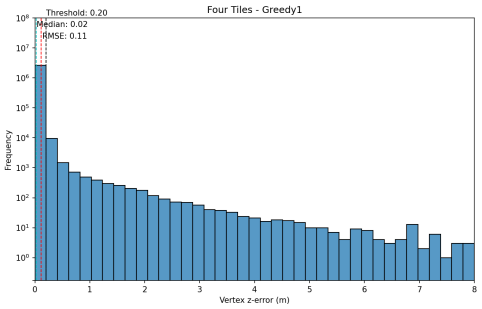
At this point the size of the TIN with no simplification became too large to be able to store it on the main drive of the PC used to run the streaming pipeline. For this reason the data is written to `/dev/null`, which means that everything written is never stored anywhere. This means that the pipeline can become slightly faster as it does not have to deal with the I/O of a storage solution. Therefore, a comparison is made for the Two Tiles dataset between the simplification while writing to a file and simplification

when writing to `/dev/null`. The difference is 15 minutes for this dataset, where the `/dev/null` version achieves a time taken of 2h35m compared to the 2h50m while storing the file. For all results from Four Tiles and up it should be noted that this `/dev/null` time taken is used to provide an indication of how long this triangulation would take, but it is not a true comparison to previous time taken measurements.

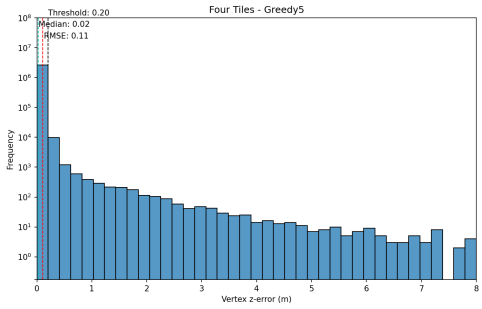
Furthermore, because data is no longer stored, the number triangles and vertices from the result cannot be reported. The data presented in Table 4.6 is therefore the number of vertices seen in Table 4.1. The number of triangles can be assumed to be approximately two times the number of vertices, which is a property seen in all of the prior results. Considering that the non-simplified versions of the data is already too large to fit in the free space of the computer used for these tests highlights the importance of having a way to effectively simplify massive datasets.



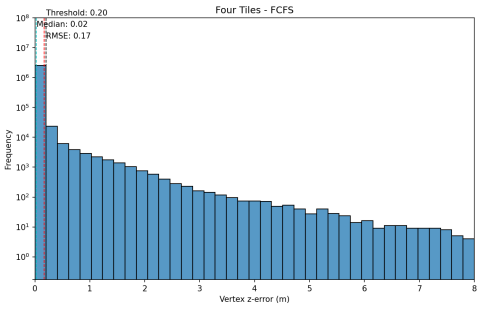
(a) Rand10



(b) Greedy1



(c) Greedy5



(d) FCFS

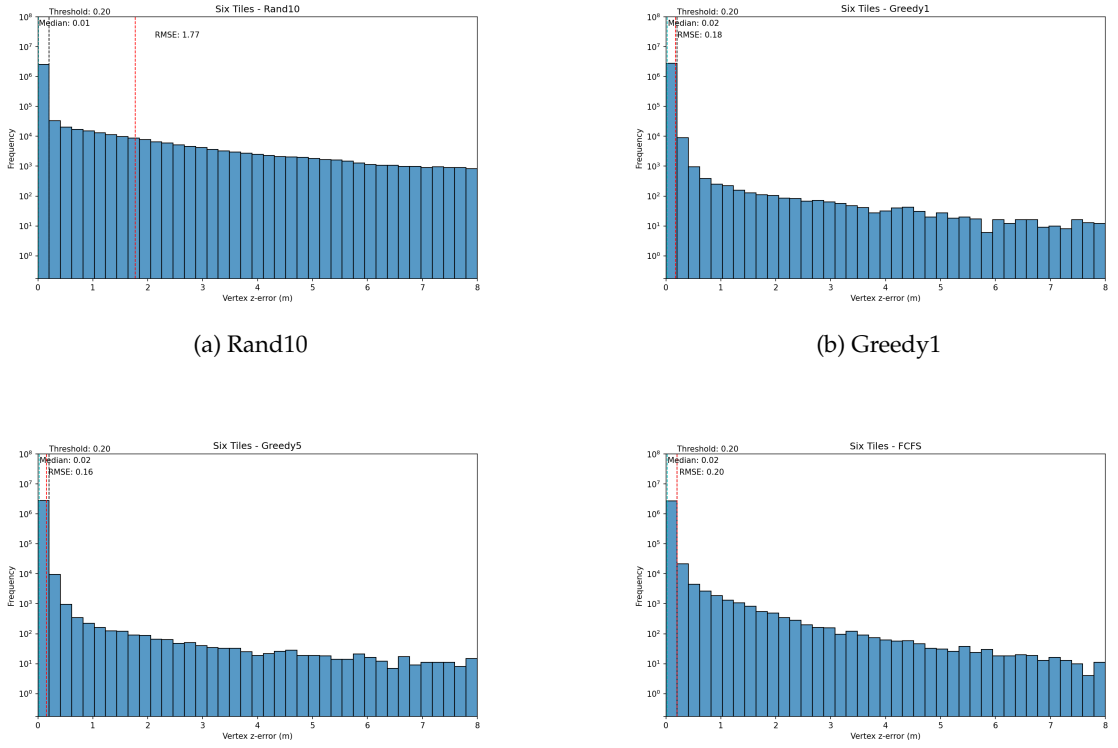
Figure 4.16: Four Tiles: Error histograms showing the distribution of error for each method.

4.4.3 Full-Scale

Table 4.7: Six Tiles: Results of the simplification methods.

	time taken	vertices	triangles	RMSE	max error (m)	max memory (MB)
No Simplification	10h19m[(1)]	1,902,652,509	-	-	-	-
Rand10	1h48m	177,564,511	355,128,936	1.75	44.2	12
Greedy1	13h31m	188,354,538	376,708,778	0.18	31.5	776
Greedy5	12h20m	227,937,404	455,874,508	0.16	32.8	804
FCFS	8h32m	165,468,046	330,935,798	0.20	35.9	859

(1) data piped to `/dev/null` instead of file



(a) Rand10

(b) Greedy1

(c) Greedy5

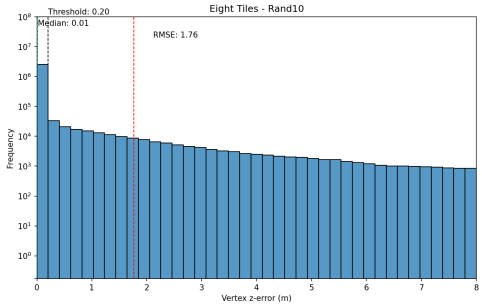
(d) FCFS

Figure 4.17: Six Tiles: Error histograms showing the distribution of error for each method.

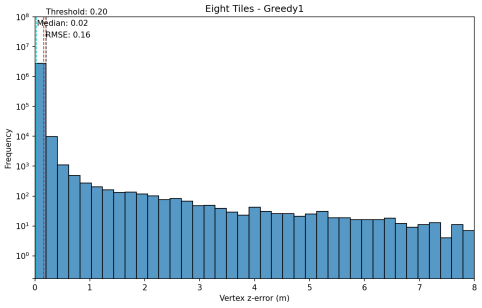
Table 4.8: Eight Tiles: Results of the simplification methods.

	time taken	vertices	triangles	RMSE	max error (m)	max memory (MB)
No Simplification	12h55m[(1)]	2,424,250,813	-	-	-	-
Rand10	2h23m	225,004,204	450,008,309	1.76	44.2	12
Greedy1	17h11m	228,154,328	456,308,291	0.16	31.5	729
Greedy5	15h23m	279,231,753	558,463,139	0.16	38.4	828
FCFS	10h50m	199,632,116	399,263,875	0.21	39.1	776

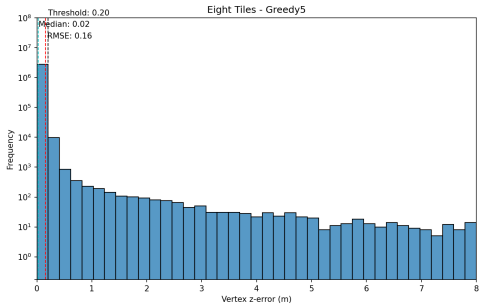
(1) data piped to `/dev/null` instead of file



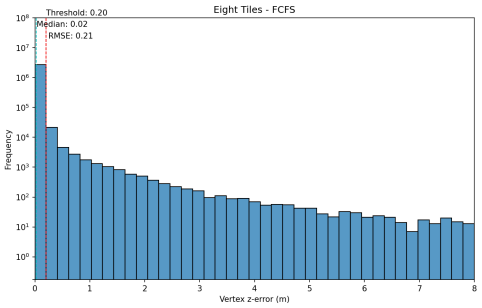
(a) Rand10



(b) Greedy1



(c) Greedy5



(d) FCFS

Figure 4.18: Eight Tiles: Error histograms showing the distribution of error for each method.

4.4.4 Comparison of Results to other Methods

The following section elaborates on how the results obtained from the simplification modules relate to the results available in prior research. In Chapter 2 (related work) three papers are identified that have applied simplification for massive triangulations; Isenburg et al. [2006d], Hegeman et al. [2014], and Dukai [2020]. Therefore, a comparison between the obtained results from the simplification modules and their prior research is made based on the available metrics or results presented in the research papers. Where possible, results or code are used.

The results shown in Constantin et al. [2010] prove to be difficult to use for comparison purposes. There is a table which links datasets and simplification rate to the time taken for simplification, however there is no stated RMSE for this set of results. Another table indicates the RMSE, but these appear to be different results because the simplification rates differ from those stated in the time measurement table. Due to the confusion between these tables no comparison is made against the results in Constantin et al. [2010].

Comparison of Results vs. Isenburg et al. Isenburg et al. [2006d] has shown that simplification can be applied in a streaming pipeline by simplifying TINs for use with a contour line extraction algorithm. They note that the simplification step is the most computationally heavy step, though they are still able to simplify around 100 thousand triangles per second on a Dell Inspiron laptop with a 2.13GHz mobile Pentium (2 core) processor and 1GB of RAM Isenburg et al. [2006d, p. 4]. The simplification step is placed at position C in the streaming pipeline thus simplifying triangles, as with the decimation module.

The comparison between the results of the simplification modules against Isenburg et al. [2006d] is based on the statistics mentioned in the research paper. This is because no code or results are available for their

methodologies and thus it is impossible to reproduce their research to make an equal comparison. Using the data available in the paper, Table 4.9 shows that the method created by Isenburg et al. [2006d] is outperformed in terms of number of triangles per second by almost all the methods created in this thesis, except for Greedy1. In terms of simplification percentage, the method created by Isenburg et al. [2006d] is still able to simplify more triangles, though this could be compensated by adjusting the threshold specified for each simplification method in this thesis. It should also be noted that Isenburg et al. [2006d] do not state what the RMSE of their method is, so a comparison of this error metric, or the maximum error, is not possible.

To calculate how many triangles can be simplified per second, the total number of triangles in the non-simplified result is divided by the number of seconds taken by the simplification method. This results in how many input triangles a simplification method can simplify on average.

Table 4.9: Comparison of simplification results from Single Tile to the results recorded by Isenburg et al. [2006d].

	triangles per second	simplification
Isenburg et al. [2006d]	100,000	90%
Greedy1	74,251	87%
Greedy5	104,905	84%
FCFS	127,051	88%

Comparison of Results vs. Hegeman et al. Hegeman et al. [2014], who simplify TINs using a large memory cluster in AWS, do provide some more information on the RMSE of their simplified dataset. Furthermore, the number of vertices that are simplified per second can be deduced using the total number of vertices with the time taken to simplify the dataset. In this case, the dataset consists of 3.64 billion vertices which can be simplified in 37 minutes using nine nodes in a cluster [Hegeman et al., 2014, p. 262].

A comparison between the method created by Hegeman et al. [2014] and the methods created in this thesis is seen in Table 4.10. Vertices per second is calculated by taking the total number of vertices in the dataset (343,038,929) and dividing that by the time taken by the pipeline to simplify and triangulate these vertices. The discrepancy between the two methods is very large in this scenario, with Greedy1 performing around 116x worse and Rand6 around 23x worse. However, it should be noted there appears to be a large trade-off between how many vertices are simplified and the RMSE of the result for Hegeman et al. [2014], who state a very large RMSE for the % reduction obtained, similar to the Rand6 method which comes closest in terms of performance. Furthermore, to achieve this result a total of 2196GB of RAM is used, 288 vCPU cores, 10Gbps networking, and 6400GB of SSD storage as these results are obtained on a 9-node cluster where each node has 288GB of RAM, 32 vCPU cores, and 800GB of SSD storage.

Table 4.10: Comparison of simplification results from Single Tile to the results recorded by Hegeman et al. [2014].

	vertices per second	simplification	RMSE (m)
Hegeman et al. [2014]	1,600,000	32%	0.07
Hegeman et al. [2014]	4,300,000	81%	1.90
Rand6	184,430	86%	1.44
Greedy1	37,125	87%	0.14
Greedy5	52,452	84%	0.14
FCFS	63,526	88%	0.18

Comparison of Results vs. Dukai. The final comparison is against the methodology created by Dukai [2020]. This simplified TIN only uses points classified as ground, which is different from all results obtained within this thesis so far. Therefore, the simplification methods created for this thesis are also re-run with only ground points to ensure a fair comparison can be made. The dataset used to compare the amount of simplification and the RMSE between these methods is seen in Figure 4.19. The results of this comparison are shown in Table 4.11.

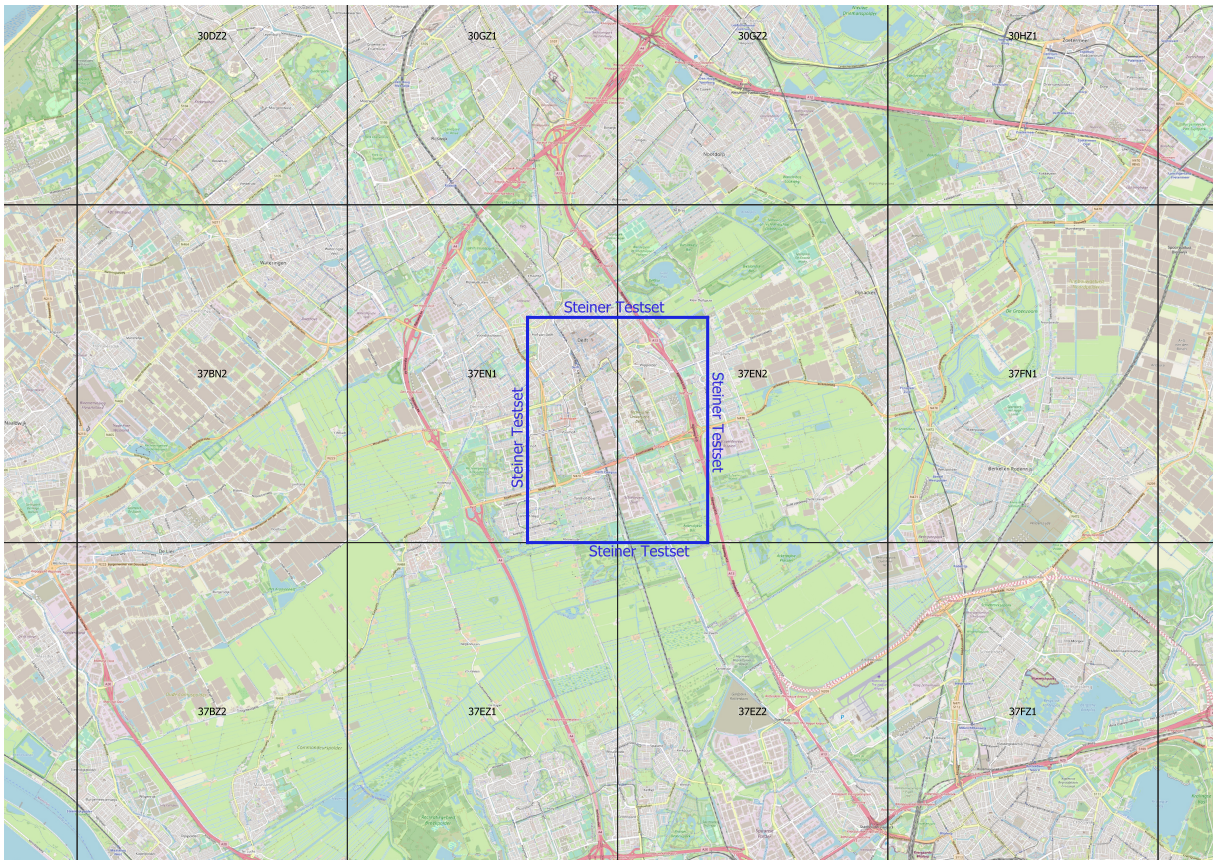


Figure 4.19: Dataset used for the comparison with Dukai [2020].

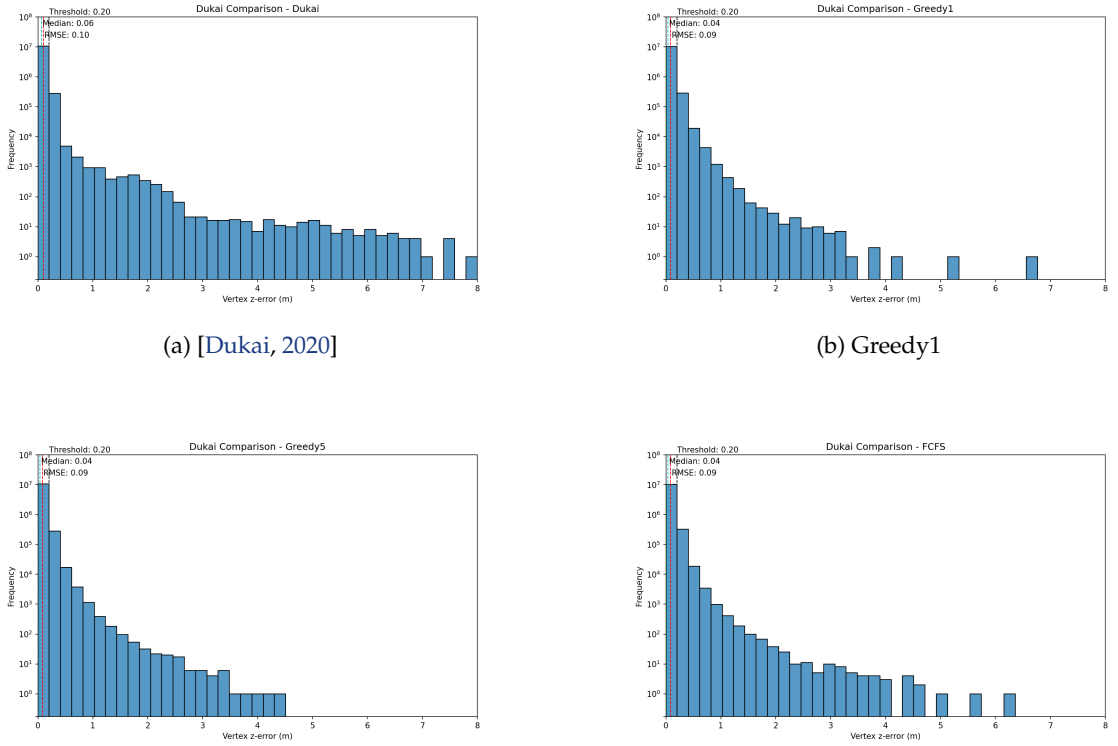
Dukai [2020] uses 3dfier [Ledoux et al., 2021b], to simplify each sub-tile using TIN simplification to a maximum threshold of 0.3m. TIN simplification is refinement while recalculating the point of maximum error after every iteration, as defined by Garland and Heckbert [1997b]. Because the 0.3m threshold differs from the 0.2m that has been used for all the previous results, each simplification method is re-run using the same threshold to ensure a fair comparison. The results show that the method used by Dukai [2020] performs worse in terms of accuracy as the RMSE of 0.10m and max error of 8.9m is higher then the 0.09m RMSE and all max errors of my simplification methods (6.7m, 4.4m, 6.3m). Histograms of these results can be seen in Figure 4.20. The histograms show that the spread of vertex errors is also lower for my methods. However, because there is no mention of the processing time taken to produce the dataset, it is difficult to say how the methods compare in terms of speed.

The advantage that Dukai [2020] achieves compared to my methods is that there is more simplification, achieving 99.7% of vertices removed compared to the next best method (FCFS) which achieves 99.3% simplification. Greedy1 and Greedy5 achieve a 98.5% and 97.6% simplification, respectively. Considering a 4 billion vertex dataset, such as the Eight Tiles dataset, Dukai [2020] would keep only 12 million vertices. FCFS would keep more than double that amount with 28 million vertices (16 million more), Greedy1 follows with 60 million vertices (48 million more), and Greedy5 would keep 96 million vertices (84 million more). The difference in how many vertices are kept for such a large dataset clearly shows the importance of the minor differences in simplification percentages. In some use-cases having

Table 4.11: Comparison of Simplification Results to Dukai [2020] based on the data seen in Figure 4.19.

	simplification	RMSE (m)	max error (m)
Dukai [2020]	99.7%	0.10	8.9
Greedy1	98.5%	0.09	6.7
Greedy5	97.6%	0.09	4.4
FCFS	99.3%	0.09	6.3

as few as possible vertices may be preferable over the 0.01m increase in RMSE and marginal increase in maximum vertex error.



(a) [Dukai, 2020]

(b) Greedy1

(c) Greedy5

(d) FCFS

Figure 4.20: Comparison of results from this thesis vs. Dukai [2020]: Error histograms showing the distribution of error for each method.

4.5 Artefacts

During the simplification of the points or TINs in the streaming pipeline artefacts may occur, as shortly discussed in Section 3.4. In the following sections a number of different artefacts are discussed which occur in the final results of the various simplification algorithms. The first of these is the quadtree border artefact which has been previously highlighted in Section 3.4. After this, the artefact at building facades is shown and its occurrence is analyzed, though it is difficult to pinpoint exactly why this artefacts occurs.

4.5.1 Quadtree Border Artefacts

The possibility of quadtree border artefacts has already been discussed by showing an example of a simplified TIN that indicates a minor level of quadtree border artefacts in Section 3.4. The image shown in that section is seen in Figure 4.21, where there is an indication of vertex clustering around the edges of quadtree borders. However, when looking at the results obtained for the Eight Tiles dataset using the Greedy1 methodology, seen in Figure 4.22, it is very difficult to discern these quadtree border artefacts.

The lack of visible quadtree border artefacts indicates that the methodology used within sst to stitch each quadtree cell together works well to mitigate these artefacts. This stitching methodology relies on releasing the true borders of each quadtree cell upon connecting cells; this ensures that in practice there is a buffer with extra points around each cell. Nevertheless, no other artefacts occur because of this buffer of higher density because these points are also simplified, resulting in a uniform distribution of points along the quadtree cell borders.

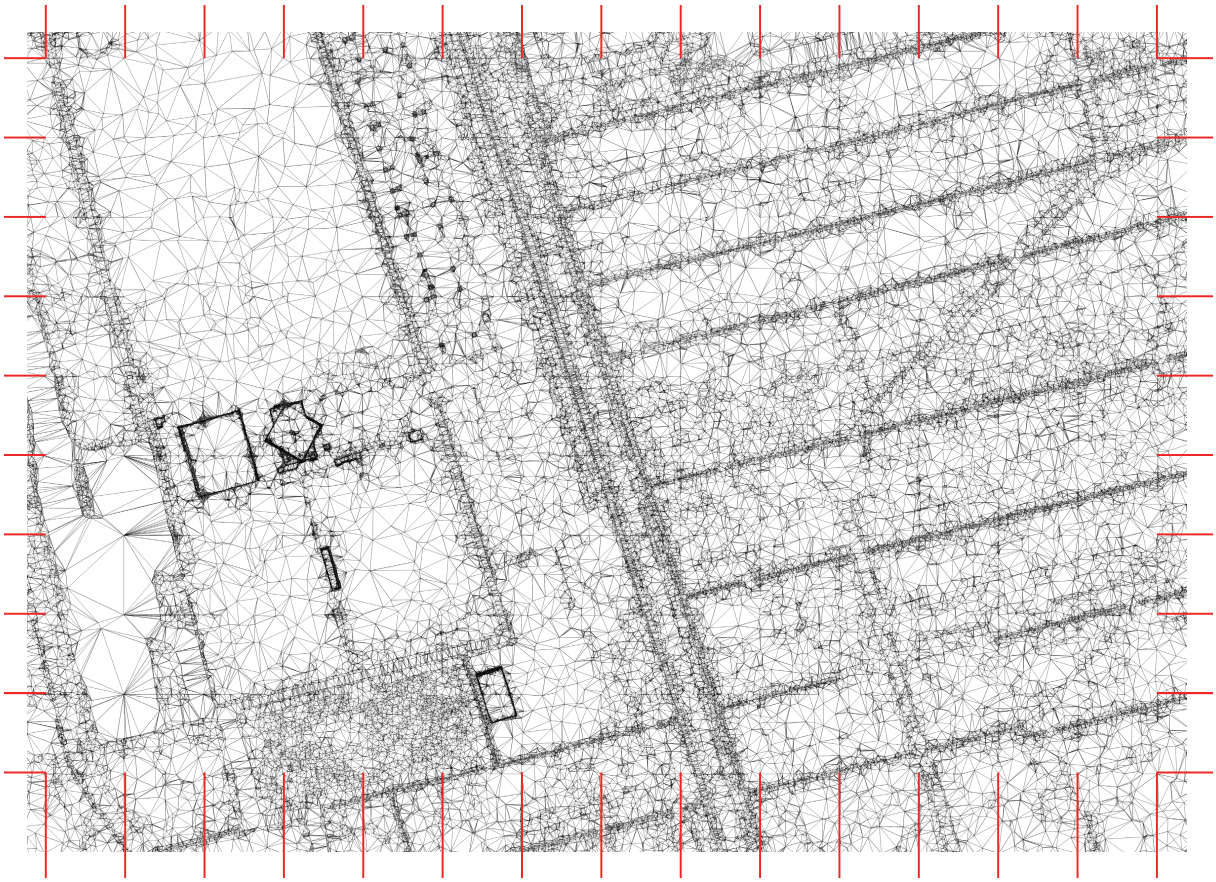


Figure 4.21: The level of visual artefacts that can occur for simplification methods implemented at position B. Red lines denote cell boundaries and are shown to assist in detecting visual artefacts.

4.5.2 Facade Artefacts

On the facades of buildings, two other clear artefacts sometimes occur which can be seen in Figure 4.23. Both of these types of artefacts occur by connecting vertices that are on the edge of a building roof to vertices on the ground that are far away from where the facade connects to the surface.

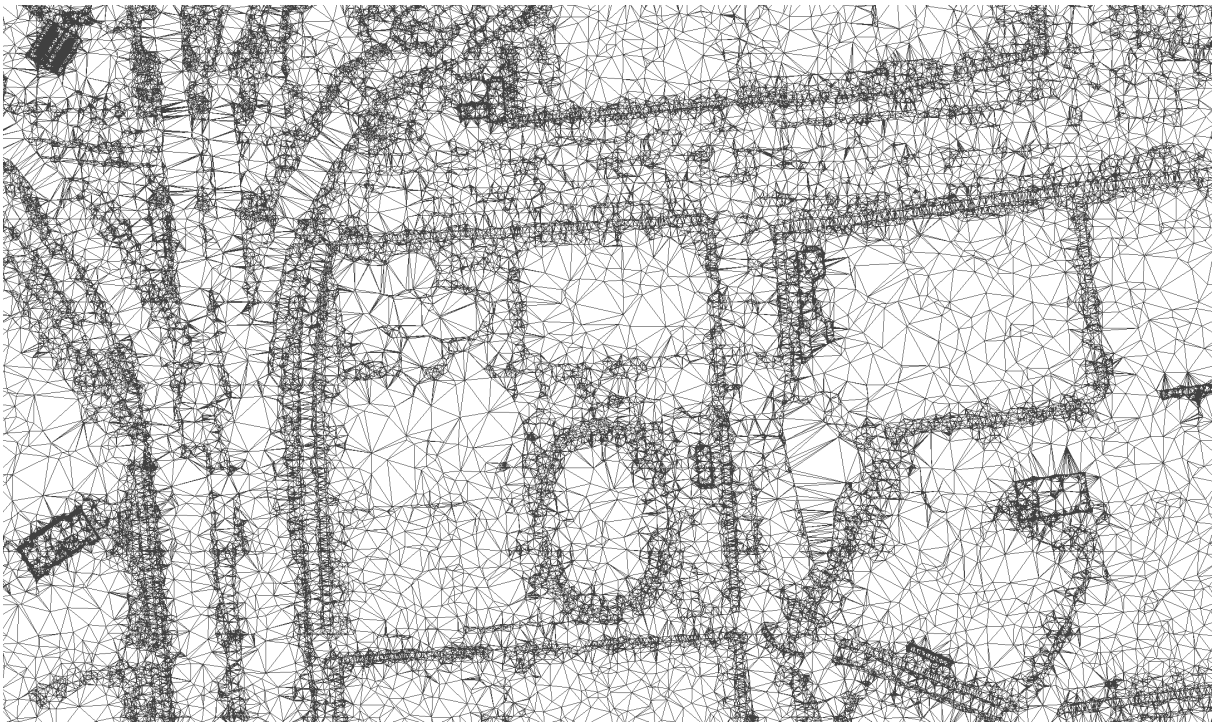
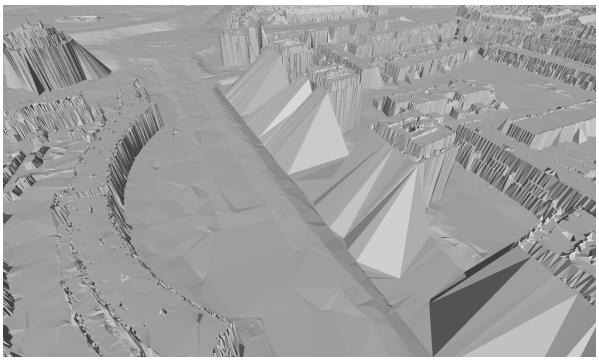
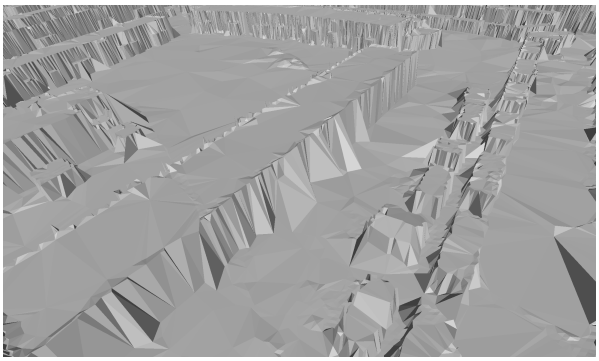


Figure 4.22: No indication of quadtree border artefacts in a small section within the Eight Tiles dataset using Greedy1.

However, the first artefact, seen in Figure 4.23a, occurs because a building is directly adjacent to a body of water and the triangles created between the roof of the building and the ground surface span this body of water. This happens because there are no LIDAR points in the water body, either due to the original dataset not containing these, or the points being filtered due to miss-classification (i.e. marked as unclassified instead of water points). The occurrence of this artefact is not avoidable within the current methodology because there is no surface approximation for building facades. Therefore, with the data that is provided to the simplification and triangulation algorithms this is the best solution that can be made, despite not being the most accurate representation of reality. Furthermore, it should be noted that the accuracy of the solution will not decrease in these cases as there are no vertices in the original dataset, thus no error will be detected and technically the provided solution is valid.



(a) Artefacts that occur between building roofs when water is adjacent to the building.



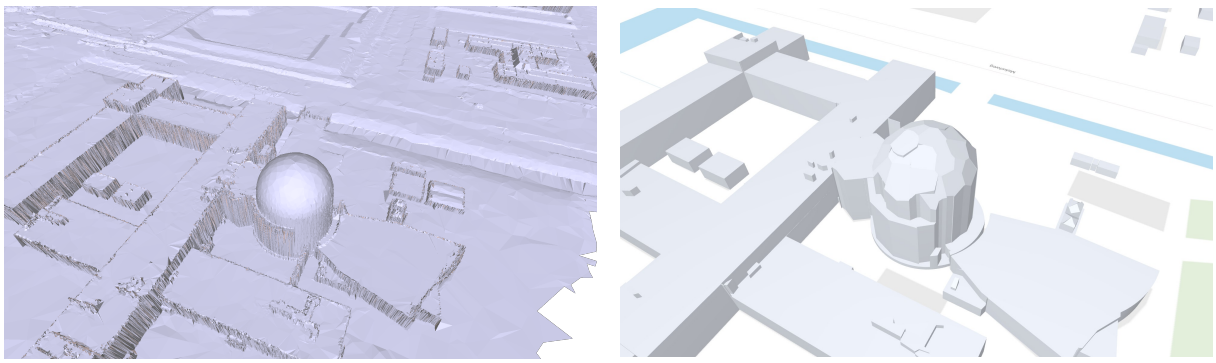
(b) Artefacts that occur between building roofs and the ground.

Figure 4.23: The two types of artefacts that occur between building roofs and the ground surface of the TIN.

In the case of the second artefact, seen in Figure 4.23b, a visually similar artefact happens. However, here there is no water body directly adjacent to the building and it is theoretically possible for the simplification and triangulation algorithms to create triangles that better represent the vertical facade. One of the reasons that this artefact may occur is that there are trees or parked cars nearby the facade of the building, which results in a similar issue to the water bodies; these points are removed by filtering and thus excluded from the triangulation. Another reason this artefact occurs is because the dataset is not entirely free from occlusion. This means that due to the angle of scanning some points behind buildings may be occluded. Often combined with the presence of a tree or other object this will lead to a similar issue with lack of vertices near the connection between the building facade and the ground surface.

To mitigate both of these artefacts which are likely to be caused by similar issues it is necessary to either use algorithms that can detect vertical surface of buildings, use the footprints of the buildings (as constraints), or by providing estimated points along the edges of water bodies (hydro-flattening). All of these solutions are out of scope for this research but will provide improved results when dealing with urban environments in triangulations in general, whether simplified or not.

Lastly, to demonstrate how well the simplification and triangulation algorithm functions in most cases, Figure 4.24a shows a complex curved building roof. The roof of the Reactor Institute in Delft in a dome shape and built as radiation barrier. Due to the dome shape some simplification methods that rely on creating straight surface will have difficulty in recreating this roof. In Figure 4.24a it can also be seen how the rest of the building is handled without introducing the same facade artefacts as described above. An example of the dome shape simplification can be seen when comparing the result of Greedy1 within the streaming simplification performed for my this to the dataset in the 3D BAG. It should be noted the the number of triangles within the 3D BAG version of this building is much less than are present in the Greedy1 simplified version.



(a) Ability of Greedy1 simplification to accurately portray complex curved building features.

(b) How simplification of the dome-shaped roof can lead to less-defined portrayal of the surface.

Figure 4.24: Comparison of the Reactor Institute dome-shaped roof between Greedy1 simplification and the 3D BAG dataset.

5 Conclusions and Discussion

This chapter discusses the results as presented in Chapter 4 and how these results answer the specified research questions raised in Section 1.2. Furthermore, the applicability of this research within the field of Geomatics is discussed as well as whether it makes sense to use a streaming pipeline instead of other methods. In the discussion there is an important question to answer related to the real-world practicality of TINs of this size. It may be desirable to create large contiguous TINs but storage will have to be done in segments or by using a DBMS, as the resulting files are still very large. Lastly, I believe the streaming geometries paradigm can be developed much further than has currently been explored in this research. Therefore, I recommend possible future work which can be done related to streaming geometries and creation of massive Delaunay TINs.

Overall, this research contributes to the geospatial world by providing something that is currently not readily available; an easy, open-source, methodology to triangulate and simplify massive point clouds when used in conjunction with sst. As far as has been found this is the first open-source simplification module available, as a binary or code, that is capable of simplifying triangulations of this size on a regular computer. Furthermore, the use of this streaming pipeline is not limited to AHN3 in any way and is expected to work on any other LIDAR dataset. This flexibility in dataset type may also open doors for researchers or institutions outside of the TU Delft and the Netherlands to simplify datasets using the simplification methods created.

The results show that my approach is able to create simplified TINs from a large number of points while maintaining a low RMSE, combined with limited memory usage. From the first result using a full AHN3 tile (Single Tile), to the Eight Tiles dataset, the memory usage for each simplification method is stable, ranging between 729MB and 955MB for the greedy refinement algorithms. It is interesting to see that the lower memory usage of these two was achieved on the Eight Tiles dataset, and the highest on the Two Tiles dataset. This means that the algorithms perform irrespective of the size of the input dataset, since memory usage shows no direct correlation with input dataset size.

5.1 Research questions

How can a seamless, simplified, Delaunay TIN for all AHN3 points be constructed using the streaming geometries paradigm?

This thesis has shown how a seamless, simplified, Delaunay TIN can be constructed using the streaming geometries paradigm by including a simplification algorithm as a step within the streaming pipeline of sst. This has been shown to function well for datasets ranging from around 200 thousand points to datasets containing around 4 billion points. For each increase of 500 million vertices in dataset size, the processing time increases by around 2h30m with the best performing method (FCFS). This is the main hurdle to overcome when scaling to the dataset size of the entire AHN3, which contains roughly 600 billion points. The methodologies presented in this thesis show that it is possible to simplify a TIN with minimal memory usage, ensuring that it is possible to construct massive Delaunay TINs on a regular PC.

Therefore, the expectation is that using the methods presented in this research it is possible to constructed a seamless, simplified, Delaunay TIN for all AHN3 points, though a lot of time is necessary to do so. All results obtained for this thesis were calculated using 8 CPU cores, assuming the same 8 core are used to calculate the entirety of the Netherlands this will take around 86 days. This is assuming every 12 hours eight tiles are processed using FCFS and there are 1374 total tiles in the AHN3

dataset. Theoretically, however, double the number of CPU cores means that approximately half the time is needed to process all this data, reducing the total time to around 43 days of non-stop processing. Switching to Greedy5 results in processing eight tiles roughly every 15 hours, or 107 days for the entire AHN3 dataset.

Using more cores will mainly benefit the simplification methods as the main pipeline cannot be parallelized to ensure data continuity between programs. However, as will be discussed in Section 5.4, the finalizer can be parallelized to improve performance in the initialization phase of the streaming pipeline. Therefore, a combination of more CPU cores, combined with a performance improvement to the finalizer, leads to a realistic scenario for using this methodology to process the entire AHN3 dataset.

How can TIN simplification be integrated into the streaming creation of a Delaunay triangulation?

TIN simplification can be applied at three positions within the streaming geometries pipeline sst; position A, B, or C (see Figure 5.1). Position C is the most disadvantageous position because a complete triangulation is created initially, prior to simplifying this triangulation. Therefore position C should never be used for simplification within a streaming geometries pipeline. Position A has the advantage of reducing the number of vertices that the finalizer has to process, increasing the processing speed of the pipeline. However, position A does not allow for multiprocessing because results depend on each other as they are not being segmented in a logical way, e.g. by releasing each finalized cell of the quadtree. This leaves position B, which has the advantage of being able to simplify data within the cells released by the finalizer, while also providing less data for the triangulator to process. With this reasoning position B is the most likely position for a simplification method to be implemented, as it allows for boosting the processing speed while working seamlessly with the existing streaming geometries infrastructure provided by sst.

Figure 5.1: Possible placement positions for a simplification module within the *sst* pipeline.

Position B also shows indications that almost all type of mesh simplification can be applied to geometries arriving at this position. This is mainly due to the isolated nature of each cell release by the triangulator, allowing for the cells to be treated as individual meshes within which a multitude of simplification operations can be done.

Which TIN simplification method produces the best results when used in a streaming pipeline?

Depending on the aim of simplification the answer to this research question can be either of the three methods that have been tested for the specified datasets. If the aim is to achieve the lowest RMSE in combination with the least amount of vertices, then Greedy1 is the answer. If the aim is to achieve a low RMSE with a trade-off between better execution times and a higher number of vertices, then Greedy5 is the answer. In the last case, if the aim is to achieve a fast result with very low vertex count and higher RMSE and max error, the FCFS method provides the best option. Each of these algorithms is capable of creating a seamless, simplified TIN for all of the tested datasets, while keeping memory usage and execution time within reason.

However, taking into considering the aim to be able to process the entirety of the AHN3 dataset it is likely that computation time is more important compared to the relatively small increase in RMSE. This is because this dataset is so large that speed is a more important metric compared to accuracy at this point. Furthermore, the FCFS method consistently has the least number of vertices and triangles of all the methods, which is preferable and will scale as the size of the dataset grows even further. For the Eight Tiles dataset of almost 2.5 billion points FCFS manages to outperform Greedy5 by around 4h30m. With the Six Tiles dataset of nearly 2 billion points this difference is around 4 hours. This suggests that for each increase of 500 million points the FCFS method is 30 minutes faster. Scaling this to 600 billion vertices easily shows how why it is preferably to choose for the FCFS method. For example, at 60 billion

points ($1/10^{th}$ of the AHN3) the difference between FCFS and Greedy5 could be 1800 hours; or 75 days of processing time.

How does the streaming creation and simplification of a Delaunay TIN perform in comparison to existing methods in terms of execution time, memory usage, and accuracy?

It is apparent that it is difficult to compare the streaming creating and simplification of Delaunay TINs to existing methods, as there are few existing methods that exist for the creation of massive TINs, nor are these methods that are commonly well documented. Nevertheless, using the information that is provided by the existing methods, a comparison is made to Isenburg et al. [2006d] and Dukai [2020]. Here, it can be seen that the method used by Isenburg et al. [2006d] is capable of processing a similar number of triangles per second and achieves a higher simplification percentage. It is unsure what type of simplification is applied by Isenburg et al. [2006d], though it is likely to be a simple form of simplification if they were able to processing 100,000 triangles per second on a single core on a laptop in 2006. Furthermore, no accuracy measurements or TIN output data is provided by Isenburg et al. [2006d] making it very difficult to validate or reproduce their results. Therefore execution time, memory usage, and accuracy cannot be relevantly compared between Isenburg et al. [2006d] and this thesis.

Luckily Dukai [2020] has provided results that can be checked and happen to be available for the same datasets as for this research. However, execution time is not measured for Dukai [2020], nor is memory usage. This leaves a comparison of the accuracy, discussed in Table 4.11. The comparison seen in that section shows that the method developed for this thesis outperforms Dukai [2020] based on the RMSE measurement of the tested dataset.

5.2 Applicability

The applicability in creating a seamless, simplified, Delaunay TIN for the entire AHN3 is quite small, mainly due to the size of the resulting files on disk. Despite simplification percentages between 80 and 90%, the Eight Tiles dataset simplified by FCFS (least resulting vertices and triangles), is still 17GB on disk as OBJ file. Considering all the tiles in the entire AHN3 dataset, storing this amount of data in a single file is impossible. Therefore it will always be necessary to split this contiguous TIN into smaller segments. This ensures that the datasets can still be loaded on a regular PC without overflowing the main memory of that PC, which is the goal of simplifying these TINs.

Using the streaming geometries paradigm for process large datasets is probably the best way to approach the issues of dealing with large datasets. Realistically, streaming of geometries is the only way to process the ever-growing geospatial datasets that are being created nowadays. Developing robust streaming ecosystems are essential to ensure that open-source geospatial data processing without the need for large servers remains possible in the future, making this research an important step in the right direction. LAStools shows the practicality of streaming data for some of their tools, such as: clipping, merging, and retrieving LAS info. Critically, however, LAStools lacks streaming for essential processes such as triangulation, simplification, and conversion to DEM or contour lines. Therefore, this research provides an initial step in the direction of filling this gap.

5.3 Discussion

Simplification within a streaming geometries pipeline is clearly possible and is capable of producing accurate results. However, Isenburg et al. [2006d] is able to perform comparably using older CPU cores, which leads to the idea that the method used by Isenburg et al. [2006d] is similar to that of Hegeman et al. [2014], and performs simple simplification in the form of thinning. Therefore, when comparing my results to that of Hegeman et al. [2014] the RMSE of my methods is better. Furthermore, it is nice to see that my methods are able to outperform the results obtained by Dukai [2020] using Ledoux et al.

[2021b]. This supports the idea that using the streaming of geometries combined with a simplification method is able to produce accurate results for massive datasets.

Practicality of massive TINs. One of the main discussion points that arises from the creation of these massive contiguous TINs is the practicality of creating these TINs in the first place. As mentioned, the FCFS Eight Tiles dataset which consists of 'only' 200 million vertices (starting from 4 billion) after simplification is around 17GB when stored in OBJ format on disk. It is highly impractical to keep these massive TINs stored as single files and it will always be necessary to split these files into smaller segments or store these in a distributed way in a database management system (DBMS). More suggestions towards how this large amount of data can be stored is found in Section 5.4.

Performance of Parallelization on Older Hardware. Another observation made is that parallelization of a process does not always function as expected, because when moving to a server with 40 cores the overall processing speed was reduced. Despite the older cores on the server machine, the expectation is that using more 4x as many cores would still increase the overall processing speed of the simplification methods. However, in practice using 32 cores compared to 8 resulted in an overall reduction in the processing speed of around 50%. Even considering the overhead in the parallelization this reduction should not be this large. In the future it is interesting to perform these simplification methods on a server with more modern CPU cores and comparing how this compares to the results obtained in my research. If a more modern server is capable of producing a more similar result to the results obtained in this thesis then that explains the 50% speed reduction. However, if this is not the case then there might be a difference in how multiprocessing is handled by Python on native Linux (installed on the server), compared to WSL2 (installed on the PC used for the results).

Creation of Simplification Algorithms. For this research most of the simplification methods were written from the ground up, which requires considerable effort in understanding methodologies and recreating specific algorithms. While educational and allowing for the tweaking of every part of the algorithm, it is likely preferable to combine simplification with more established existing methods. MAT simplification is implemented by calling the simplification program from the streaming pipeline in a separate process. This same technique can be applied to nearly any simplification method at position B. This is the case because generally the number of points within a single quadtree cell is small enough for non-streaming specific simplification methods to deal with. By using this method it is also possible to iterate through a larger number of simplification methods.

Inclusion of Building Points in Simplification. Another discussion point is that the choice has been made to keep building points when simplifying the datasets, which increases computational complexity and extends processing time by quite a lot. When comparing to the method created by Dukai [2020] this became apparent when a simplification was made of only surface points. This suggests that creating a TIN of only the surface of all AHN3 points is more doable than when including building points. Therefore, it may be interesting to use the streaming geometries paradigm to create the DSM of the entire dataset and combine this with the 3D models created by the 3D BAG to create a full representation of the Netherlands. Nevertheless, not many countries have a 3D BAG dataset containing 3D models of all buildings, thus it is good to see that my methods are capable of dealing with buildings while creating relatively few critical artefacts.

Memory Limitations of Simplification. Lastly, it is interesting to determine whether there is a limitation to how large the input dataset can be when limited in the amount of RAM that can be used. When considering only the simplification methods, the theoretical limit is based on the number of cores that can be used. The simplification methods do not store any persistent data and only keep the points from the quadtree cells that are being processes in memory while they are being simplified. The number of quadtree cells is dependent on the number of cores, where the results obtained suggest a maximum of 200MB of RAM per quadtree cell may be used at a time. Therefore, as long as a PC has 200MB of free

RAM per processing core available the simplification methods will not run out of memory. This calculation is done based on the cell size of 50m which has been used throughout this research. For larger cell sizes the amount of memory necessary will be more and can thus lead to bottlenecks.

Limiting Interval for Greedy Algorithms. Currently the interval for the greedy algorithms increments indefinitely until the triangulation is finalized. The effect of this is that at a certain point the number of points being added in each iteration is very high, which may result in the introduction of more of the small triangles. Therefore, it makes sense to limit how far this interval is allowed to increase. This would ensure that, for example, a maximum of 25 vertices is ever added in a single iteration. This is a relatively small change, but may significantly affect the visual quality of the resulting TINs.

5.4 Future work

Considering the enormous growth in volume and density of geospatial data, it is hard to ignore that streaming of geometries provides an adequate processing solution. Despite having been introduced in 2006 there have hardly been real improvements made to the overall process. Therefore, there is still quite a lot of future work that is possible with the streaming geometries paradigm.

Performance of the Finalizer. The finalizer that is part of sst currently does three passes over the dataset to determine certain required initialization parameters. When supplying multiple files as an input dataset the finalizer will pass over each file sequentially to retrieve these parameters. However, it is also possible to pass over these files in parallel, offloading each file to a separate process to be handled by a different core. This will benefit the initialization time needed by the finalizer to start releasing points to the streaming pipeline, thus increasing the overall processing time of the sst pipeline.

Storing Massive TINs. The storage of the massive TINs created in my thesis is done in a single (streaming) OBJ file which includes finalization tags as separate lines at the moment, which is far from practical due to the file size. Therefore, it is necessary to use a more practical method for storing massive TINs to allow them to be more usable, better accessible, and highly compressed. One of the most logical approaches to storing these massive TINs is to use a DBMS, or more specifically the approach suggested by Ledoux [2009]. This method relies on storing vertices with their x, y, and z coordinate, as well as their star. The star provides information on the neighboring vertices that comprise the triangles around the vertex. Another advantage of using a DBMS is that it allows for accessing of smaller sections of data from the large dataset. An example of this is drawing a bounding box and only requesting the TIN that is contained within that bounding box. From a DBMS this information can be retrieved in a reasonably easy fashion.

A useful addition to create as a module within sst's streaming geometries ecosystem is to be able to store the TIN data directly to a DBMS. This should be fairly straightforward as long as the triangulator has the vertex position and its neighbors. Using that data another module can write this output to the DBMS in the format suggested by Ledoux [2009].

Expanding FCFS with more processing loops. Currently FCFS features two processing loops, a coarse loop with a threshold of 2m and a fine loop with a threshold of 0.2m. It would be interesting to examine what the effect would be of adding more processing loops. For example, adding an even coarser threshold prior to the 2m threshold to remove only the largest vertex errors. Or adding another loop between the coarse and fine thresholds that can be used to reduce the number of small triangles that are still created by the current FCFS configuration. Another option is to combine both these additional loops to have four increasingly fine thresholds that may theoretically increase the accuracy of the results produced by FCFS, while only marginally increasing the processing time.

Simplification with Quadric Error. Based on the related work assessed for this thesis, the quadric error method with edge contraction has the potential to create TINs with a low RMSE while also being reasonably fast to compute. For this thesis the approach is not used as it results in moving vertices from their original location. However, considering that a simplified TIN is made to represent the surface created by a point cloud with as few vertices as possible, using quadric error edge contraction may prove to provide better results. Since edge contraction relies on moving vertices, it allows for better fine-tuning of the output TIN, theoretically reducing the number of vertices while keeping the RMSE within bounds. One of the complications with edge contraction may occur around cell borders, where more severe artefacts may occur as moving of vertices is not possible there. However, it is still relevant to assess the practicality of this methodology.

Determining which points are within a triangle. Being able to determine which points lie within a triangle will benefit the simplification methods but is currently not possible within *Startin*, the triangulation library used within the simplification methods. The benefit lies in being able to recalculate the errors of relevant vertices, as opposed to recalculating all vertex errors. It is difficult to say how this can be achieved without increasing the computational complexity of Startin. One of the options is to use the coordinates of a triangle to determine which points are within the bounding box to subsequently do a point in triangle test for each point to link the triangle and point together. However, this method will be very slow as it relies on using a point in triangle test. Another method may be to track each point within a triangle from the start, then when inserting a new point determining whether the points that are within that triangle are on the left or right of the created edge. The added complexity here is related to having to retrieve the points from an adjacent triangle when a flip action is done after inserting a new point. However, it sounds like a doable approach and can be explored as future work to greatly improve the performance of the simplification methods.

Creation of more modules for sst. Besides the creation of a module to store the resulting TIN to a DBMS, other modules should also be explored which can enrich the streaming geometries ecosystem. Examples of these modules are: contour line extraction, DTM/DSM creation, noise modelling, and watershed calculation. The two latter problems would require starting with a completed TIN before being able to use streaming to perform these calculations. Creating these modules allows for the creation of more types of data from these massive datasets and supports the ability to create contiguous datasets.

Streaming of Real-Time Data. A possible application that would be interesting to see is whether real-time (sensor) data can also be streaming into the streaming geometries pipeline. This can become interesting when doing calculations related to determining flood risk of certain areas. By using the sensor data that is pushed into the streaming pipeline the calculation for possible risk areas can be kept up to date on a large scale. This would allow for larger target areas to be addressed compared to focusing on a specific high-risk areas.

A Reproducibility self-assessment

A.1 Marks for each of the criteria

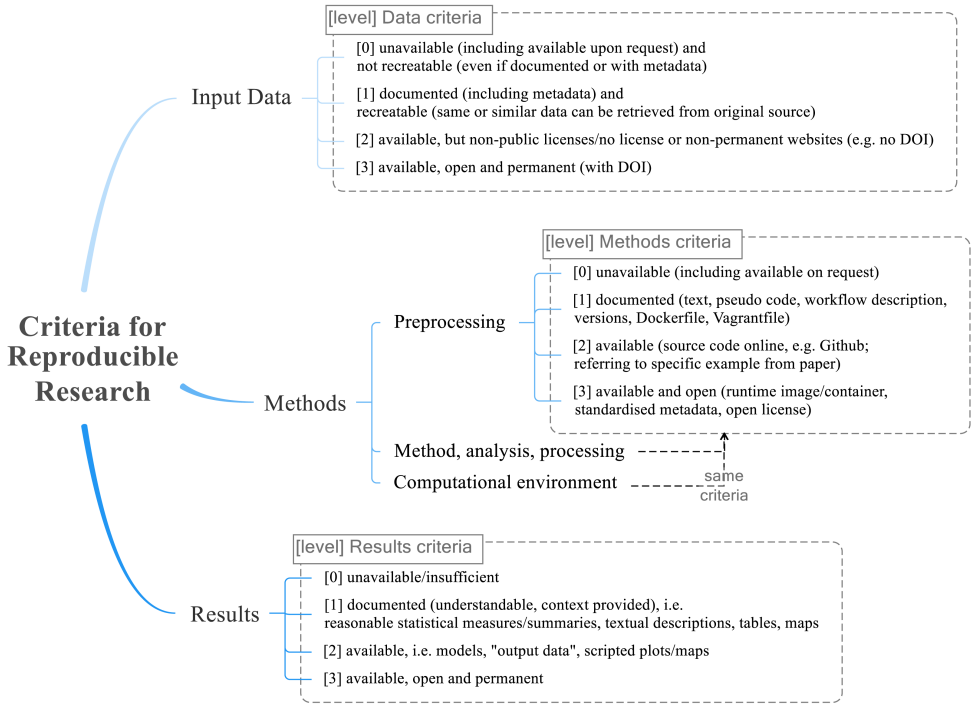


Figure A.1: Reproducibility criteria to be assessed.

Grade/evaluate yourself for the 5 criteria (giving 0/1/2/3 for each):

1. input data: **3**
2. preprocessing: **3**
3. methods: **3**
4. computational environment: **2**
5. results: **1**

A.2 Self-reflection

In theory everything is available to reproduce the contents of this thesis in the GitHub repo `https://github.com/mdjong1/simpliPy` combined with `https://github.com/hugoledoux/sst`. This former repository also includes an explanation on how to use the *sst* pipeline and how to incorporate the various simplification methods into the pipeline. An attempt has been made here to make the scripts as friendly as possible by including command line arguments for the parameters allowing the simplification methods to be treated as a black box.

The input data, AHN3, is publicly available on https://downloads.pdok.nl/ahn3-downloadpage/ and can be downloaded by anyone free of charge. Further preprocessing is not necessary as the finalizer in *sst* has been modified to include a command line argument that allows for choosing of which classes to keep. This prevents having to preprocess the point cloud with a tool such as las2las.

The computational environment is not complex and a *requirements.txt* is included where necessary for the Python scripts. The compilation of the *sst* and installation of the required Python packages may be the hardest part, though these should be easily manageable for most people interested in this research.

For the results I feel like I am lacking slightly, as the results are only available within this thesis and on my personal computer and do not live in a repository or website in an easily accessible way. This is partially due to the size of the results (TINs of multiple GBs) and partially because there are a lot of different methods run for a lot of different datasets, making uploading of all these results separately a task.

Overall I believe the research should be fairly reproduce-able with the available repositories, explanations within those repositories, and freedom granted in the form of an MIT license on all the methods created. Furthermore, some extra care is taken to write clean code with comments where necessary to allow for further development to take place in the future.

Bibliography

Agarwal, P. K., Arge, L., and Yi, K. (2005). I/O-efficient construction of constrained delaunay triangulations. *Lecture Notes in Computer Science*, 3669:355–366.

AHN (2020). AHN Kwaliteitsbeschrijving. https://www.ahn.nl/kwaliteitsbeschrijving.

Amenta, N. and Choi, S. (2008). Voronoi methods for 3d medial axis approximation. In *Computational Imaging and Vision*, pages 223–239. Springer Netherlands.

Buchin, K. and Mulzer, W. (2011). Delaunay triangulations in O(sort(n)) time and more. *Journal of the ACM*, 58(2):1–27.

Constantin, C., Brown, S., and Snoeyink, J. (2010). Implementing streaming simplification for large labeled meshes. *2010 Proceedings of the 12th Workshop on Algorithm Engineering and Experiments, ALENEX 2010*, pages 149–158.

Dukai, B. (2020). Full AHN3 TIN using Steiner Points. http://godzilla.bk.tudelft.nl/tin/gpkg/.

Funke, D. and Sanders, P. (2017). Parallel d-D delaunay triangulations in shared and distributed memory. *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 0:207–217.

Funke, D., Sanders, P., and Winkler, V. (2019). Load-Balancing for Parallel Delaunay Triangulations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11725 LNCS:156–169.

Garland, M. and Heckbert, P. (1997a). Fast triangular approximation of terrains and height fields. *Submitted for publication*, (October 1999):1–19.

Garland, M. and Heckbert, P. S. (1997b). Surface simplification using quadric error metrics. *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1997*, pages 209–216.

Hegeman, J. W., Sardeshmukh, V. B., Sugumaran, R., and Armstrong, M. P. (2014). Distributed LiDAR data processing in a high-memory cloud-computing environment. *Annals of GIS*, 20(4):255–264.

Isenburg, M. and Gumhold, S. (2003). Out-of-core compression for gigantic polygon meshes. *ACM SIGGRAPH 2003 Papers, SIGGRAPH '03*, pages 935–942.

Isenburg, M. and Lindstrom, P. (2006). Streaming Meshes. pages 231–238.

Isenburg, M., Lindstrom, P., Gumhold, S., and Shewchuk, J. (2006a). Streaming compression of tetrahedral volume meshes. *Proceedings - Graphics Interface*, 2006:115–121.

Isenburg, M., Liu, Y., Shewchuk, J., and Snoeyink, J. (2006b). Streaming computation of delaunay triangulations. *ACM Transactions on Graphics*, 25(3):1049–1056.

Isenburg, M., Liu, Y., Shewchuk, J., Snoeyink, J., and Thirion, T. (2006c). Generating raster DEM from mass points via TIN streaming. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4197 LNCS:186–198.

Isenburg, M., Liu, Y., and Snoeyink, J. (2006d). Streaming Extraction of Elevation Contours from LIDAR Points. *Citeseer*, (1):6.

Ledoux, H. (2009). Storage and analysis of massive tins in a dbms.

Ledoux, H. (2020). Streaming Startin. https://github.com/hugoledoux/sst.

Ledoux, H., Arroyo Ohori, K., and Peters, R. (2021a). *Terrainbook*.

Ledoux, H., Biljecki, F., Dukai, B., Kumar, K., Peters, R., Stoter, J., and Commandeur, T. (2021b). 3dfier: automatic reconstruction of 3d city models. *Journal of Open Source Software*, 6(57):2866.

Lee, J. (1989). A drop heuristic conversion method for extracting irregular networks for digital elevation models. In *Proceedings GIS/LIS '89*, pages 30–39, Orlando, USA.

Peters, R. (2018a). Geographical point cloud modelling with the 3D medial axis transform.

Peters, R. (2018b). masbcpp. https://github.com/tudelft3d/masbcpp/tree/kdtree2.

Ruppert, J. (1995). A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 18(3):548–585.

URI (2021). TIN Model - University of Rhode Island. https://www.edc.uri.edu/nrs/classes/NRS409509/Lectures/8Models/AnalysisGraphics/TIN_Model.gif.

van Rijssel, L., Dinklo, C., Prusti, M., Giannelli, D., and Hobeika, N. (2020). 3D noise simulation. http://resolver.tudelft.nl/uuid:9e83e3c1-0d7b-4026-a34c-2fbb61aaec2c.

Wu, H., Guan, X., and Gong, J. (2011). ParaStream: A parallel streaming Delaunay triangulation algorithm for LiDAR points on multicore architectures. *Computers and Geosciences*, 37(9):1355–1363.

Yao, X. and Li, G. (2018). Big spatial vector data management: a review. *Big Earth Data*, 2(1):108–129.

Colophon

This document was typeset using LaTeX, using the KOMA-Script class `scrbook`. The main font is Palatino.