Quantized Mesh Generation:

Software for Transforming Altitude Data into Tiled Representations

Max Lundström

Master's Thesis
Supervisor: Jan Westerholm
Department of Information Technologies
Faculty of Science and Engineering
Åbo Akademi University
Student Number: 41247
May 2023

Abstract

This thesis presents the design and implementation of a software tool for Navielektro Ky that generates QuantizedMesh tiles from elevation data. While maps are incredibly useful for geographical analysis and visualization, achieving desirable results when utilizing them for more precise analysis and visualization can prove challenging. One solution to this problem is the use of digital elevation models (DEMs), which capture the topographical structure of the Earth's surface. DEMs can be used to create highly accurate and detailed maps, but unfortunately, most of them are distributed in formats that themselves are not especially flexible, and require further processing for specific use cases.

QuantizedMesh tiles provide a solution to this problem by dividing the Earth's surface into square tiles of terrain of varying levels of detail. These tiles are compressed and can be dynamically loaded and replaced during runtime, which proves especially useful in cloud-based applications, as the users only load tiles at the location and the level of detail they are interested in, resulting in lower bandwidth requirements. The use of QuantizedMesh tiles is increasingly popular in various applications, such as virtual globes, online maps, and flight simulators.

The algorithm presented in this thesis used to generate the QuantizedMesh tiles involves first creating a high-detail level of tiles and then sampling data from that layer to create the lower-detail levels. It begins with segmenting input elevation data into tile map service (TMS) grid cells. The data in these cells are then used to create QuantizedMesh tiles. In the case that the data for one TMS cell is sourced from multiple sources of elevation data, the resulting tile automatically merges both datasets to complete the tile. The resulting QuantizedMesh tiles are optimized into triangulated irregular networks (TIN) to reduce the number of redundant data points while retaining the shape of the original mesh. The tile creation algorithm scales linearly in terms of the amount of input data in relation to computing time, and allows for processing large datasets without memory issues.

The development of this tool was necessary as there are hardly any publicly available options for generating QuantizedMesh tiles that can handle the potential scale of computation required by the project stakeholders. The tiles produced by the tool can be used for various different use cases, such as line of sight calculations and terrain visualization.

Contents

1 Introduction **4**

2 Map projections and coordinate systems **6**

2.1 Overview . . . 6

2.2 Projection . . . 6

2.3 Projection types . . . 8

2.4 Coordinate systems in cartography . . . 9

2.4.1 Earth-Centered, Earth-Fixed coordinate system . . . 9

2.4.2 Ellipsoidal coordinate system . . . 10

2.4.3 Universal Transverse Mercator coordinate system . . . 11

2.4.4 Tile Map Service . . . 11

3 Quantized Mesh **13**

3.1 Overview . . . 13

3.2 Header . . . 14

3.3 Vertex Data . . . 15

3.4 Index Data . . . 16

3.5 Edge Indices . . . 17

3.6 Extensions . . . 19

4 Project Introduction **20**

4.1 Project overview . . . 20

4.2 Design . . . 20

4.2.1 Creation of tiles . . . 22

4.3 Tools . . . 23

5 Project Implementation **25**

5.1 QuantizedMesh I/O . . . 25

5.2 Input data . . . 26

5.3 Data segmentation . . . 28

5.3.1 First look . . . 28

5.3.2 The problem with gaps . . . 28

5.3.3 Algorithm extension . . . 29

5.4 Mesh creation . . . 32

5.4.1 Triangulation . . . 32

5.4.2 Edge preservation 33
5.4.3 Mesh merging 36
5.4.4 Mesh sampling 38
5.5 Mesh optimization (TIN) 40

6 Algorithm scalability analysis 44
6.1 Overview 44
6.2 Data collection 44
6.3 Results 45

7 Discussion 47

8 Conclusion 49

9 Generering av kvantiserade nät - Svensk sammanfattning 51

Chapter 1

Introduction

Monitoring our surroundings, and keeping track of their evolution over time, is a fundamental part of modern society. It enables us to collectively make informed decisions regarding logistics, development of industry, national security, etc. The process of surveying the local land can in theory be performed by any individual group or organization. However, the important assignment of periodically conducting nationwide land surveys is generally entrusted to a governmental land surveying agency. In the United States, this agency is the *United States Geological Survey* (USGS), while in Finland, the equivalent organization would be the *National Land Survey of Finland* (NLS) or *Geolocial Survey of Finland* (GTK). These agencies provide the public with readily available information regarding their surroundings. This information includes maps, spatial data, property lines, etc. Additionally, aerospace agencies such as the *European Space Agency* (ESA) publish data gathered by their satellites, such as imagery of the Earth as well as various other observations and measurements.

While aforementioned geospatial data is generally readily available, it is usually distributed in only a few different formats, none of which might be appropriate for some particular use cases. Thus, it is up to individual organizations themselves to further transform the source data into a format more suitable for their particular use case. The objective of this thesis is to demonstrate how public geospatial data can be transformed into a tile-based mesh format for the purpose of visualization and topographical analysis. This project was developed for Navielektro Ky, a Finnish company specialized in surveillance and communication systems for civilian as well as military purposes. The project involves transforming, segmenting, and utilizing elevation data published by the NLS to create tiles of optimized 3D terrain meshes of varying levels of detail. This thesis will mainly cover the findings made during the development of said project, as well as offer solutions pertaining to how various hurdles during design and development were addressed. Additionally, this thesis covers the background knowledge required to tackle a problem of this nature.

The target output format of the data is referred to as *"QuantizedMesh"*, which is a tile based mesh format optimized for a small file size. While the QuantizedMesh format has become somewhat established, it still lacks well-supported open source libraries that aid in the reading and writing of QuantizedMesh tiles. Although there are a few open source tools that can produce QuantizedMesh tiles, none is capable of operating with the massive amounts of data (areas covering countries/continents) that are required for this particular project. While it might have been possible to modify some available tool to fit the project requirements, it was deemed as more beneficial to implement the QuantizedMesh format, as well as the entire production pipeline, from scratch. The main reason behind this is that it allows for much more control over the pipeline, as well making it easier to integrate the project into existing environments.

The main issue that needs to be solved is how to manage dense elevation data covering large areas such as countries and continents quickly and without running out of memory on a standard workstation (approx. 16 GB of memory). Most available open source tools can only handle small amounts of data, are difficult to operate, and do not support the file format or coordinate system of our elevation data. The project demonstrated in this thesis can handle virtually unlimited amounts of data in a manner that does not consume memory excessively. Additionally, the quality of the resulting QuantizedMesh tiles is kept high, while the file size is minimized.

The second chapter of the thesis will establish the underlying fundamentals of coordinate systems, map projections, etc., needed to understand why different challenges regarding the project occurred and how they were solved. After this, chapter three provides an insight into the QuantizedMesh format. The chapter aims to provide an adequate understanding of the format in order for the reader to better understand why certain features of the pipeline were necessary. Chapter four covers the design and methodology behind the project. It provides a brief overview of the various design-related problems that occurred during development and their respective solutions. In chapter five, the implementation of the project is explained in detail. It aims to provide a more detailed view of the aforementioned problems and solutions, as well as cover the details of how the design was implemented. Finally, chapter six and seven complete the thesis with a discussion about the findings throughout the project as well as a conclusion, summarizing the main points of interest and providing an overall impression of the results.

Chapter 2

Map projections and coordinate systems

2.1 Overview

Maps are essential tools when it comes to visualizing and analyzing our surroundings and have been a vital part of any explorer's toolkit for thousands of years. The impact that maps have had on the world throughout history should not be understated. Maps have aided in everything from navigation at sea to meeting an acquaintance who lives a few towns away or planning intercontinental voyages to countries on the other side of the Earth. In this day and age, with the widespread accessibility of smartphones, maps are used more than ever before.

Most people employ the use of standard flat rectangular maps, as those are most readily available from online map providers. However, as the shape of the Earth is an oblate spheroid and not a flat rectangle, there are great difficulties when deciding how its surface should be visualized. Nevertheless, if visualizing the earth on a flat surface is difficult, why not always use globes that can more accurately display the actual shape of the earth instead? The main reason for this is due to that when using maps for every-day navigation, most people are only interested in smaller areas. Using a globe for small scale visualization is usually not feasible, as the globe would have to be very large for anyone to make out any small features. Additionally, rectangular images or data are much simpler formats to process on computers, which is how most people interface with maps.

2.2 Projection

In cartography, projection is the "systematic representation on a flat surface of features of a curved surface, as that of the Earth"[11]. As previously mentioned, this can be quite a complicated process. Nevertheless, what is actually so complicated regarding the projection of the Earth's surface onto a flat plane?

The first reason is distortion. For every type of map projection, there is always some compromise. A flat map can only be accurate at certain locations. This means that there is not one map projection that could be used for every instance, thus there is a need for multiple types of projections depending on the type of interest[9].

The second reason is due to the Earth's irregular shape. The Earth is not a perfect sphere, instead it is more akin to an oblate spheroid, where the radius of the equator is greater than the polar radius. It is believed that this phenomenon is caused by the rotation of the Earth. Other irregularities of the shape are due to the gravitational differences across the Earth as it varies in density. How this variation affects the Earth is defined by the *geoid* (Fig. 1). The geoid is a reference of the true surface of the Earth.

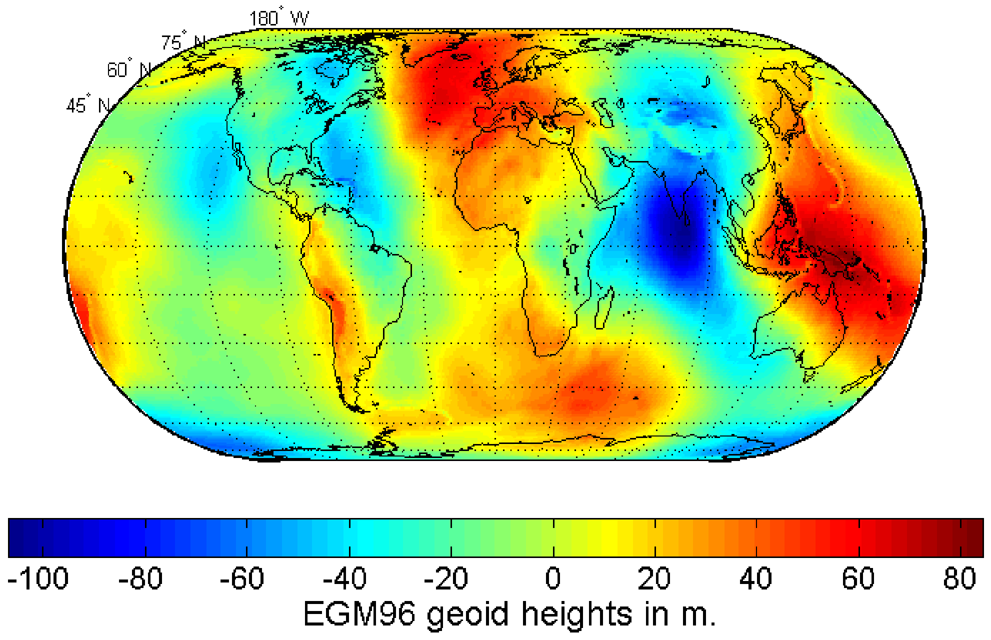


Figure 1. A reference geoid of the Earth, released in 1996 by NASA[10].
In the regions with negative height the gravity is stronger, while in regions with positive height the gravity is weaker.

The final reason is due to irregular measurements. Whether it is due to some inconsistency in the measuring device or in the assumptions that are made when analyzing the measurement, the methods used virtually always introduce some form of error. Additionally, some measurements might become inaccurate over time due to plate tectonics and changes in the Earth's crust.

2.3 Projection types

As mentioned in the previous section, a rectangular map can only be accurate in specific areas. Where the map is accurate depends on what type of projection was used for its creation. Some commonly used types of map projections are cylindrical, conical, and azimuthal.

A cylindrical projection places the Earth at the center of a cylinder and stretches the surface of the Earth onto the surface of the cylinder. Altering the orientation of the cylinder in relation to the Earth yields maps with differing qualities. In addition to projecting the Earth on a cylinder, the surface of the cylinder then has to be truncated in the Y-axis as the distortion becomes too great. The poles have to be excluded, as they are parallel to the surface of the cylinder (Fig. 2). A vertical cylindrical projection is most accurate around the equator while a horizontal cylindrical map is most accurate around the central meridian. In the case of a vertical cylindrical projection, distortion occurs when moving away from the equator. The closer to the poles that features are located, the more distorted they will be. Therefore, this type of projection is not suitable for locations such as any of the Nordic countries, which are located far up north away from the equator. A common vertical cylindrical map projection is the *Mercator* projection, widely used for maritime purposes (Fig. 2). Web Mercator[4], a variation of the standard Mercator projection, is extensively used in digital mapping software such as "Google Maps".

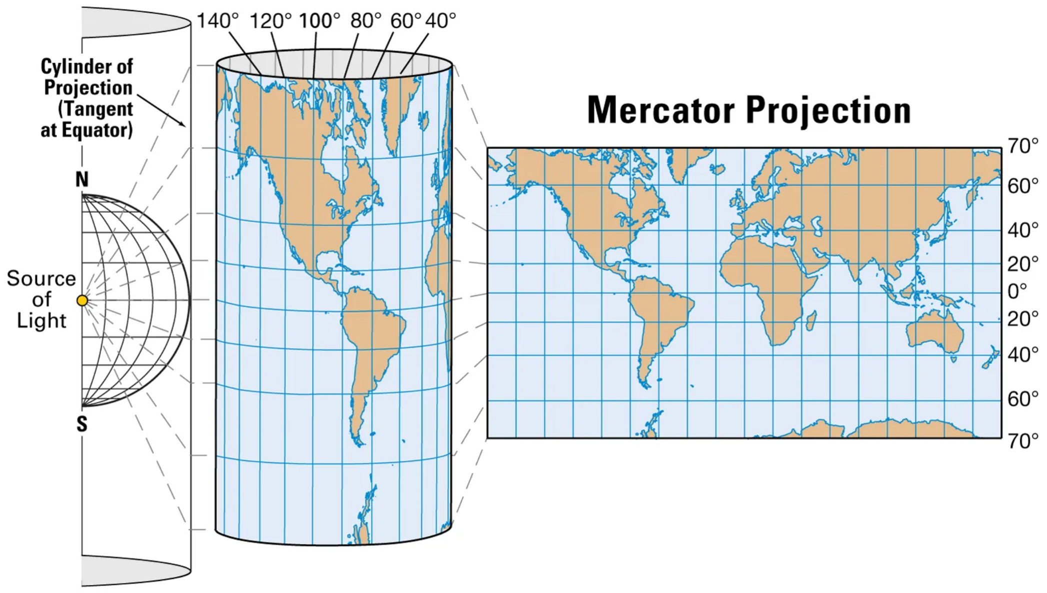


Figure 2. Mercator projection[12].
Projecting Earth surface onto surface of cylinder.

2.4 Coordinate systems in cartography

To understand how coordinate systems in cartography work, it is vital to know the difference between a geographic coordinate system (GCS), and a projected coordinate system (PCS). In essence, a GCS is what defines a model of the Earth, while a PCS projects a GCS onto a flat surface. A PCS is created by using a projection such as Mercator or Robinson in conjunction with a unit type (and other various parameters depending on the PCS) to map the ellipsoid to a plane, as described in the previous section[5]. The different types of coordinate systems relevant to this thesis are the following:

- Earth-Centered, Earth-Fixed coordinate system
- Ellipsoidal coordinate system
- Universal Transverse Mercator coordinate system
- Tile Map Service

2.4.1 Earth-Centered, Earth-Fixed coordinate system

Earth-Centered, Earth-Fixed (ECEF) is a 3D geocentric coordinate system that places the center of the earth (or any reference ellipsoid) at the origin. It is based on the standard cartesian coordinate system most commonly used in mathematics[6]. In the ECEF system, the X-axis passes through the point where the equator and the prime meridian intersect (Fig. 3). The prime meridian is a plane that splits the globe into a western and an eastern hemisphere. On the Earth, it passes through Greenwich, London, England and, along with the equator, functions as a reference line in many geographic coordinate systems. The Z-axis in the ECEF system passes through the north and south pole, while the Y-axis is perpendicular to both the X- and Z-axis, according to a right-handed coordinate system. An ECEF coordinate simply defines a point in 3D space in relation to the center of the Earth.

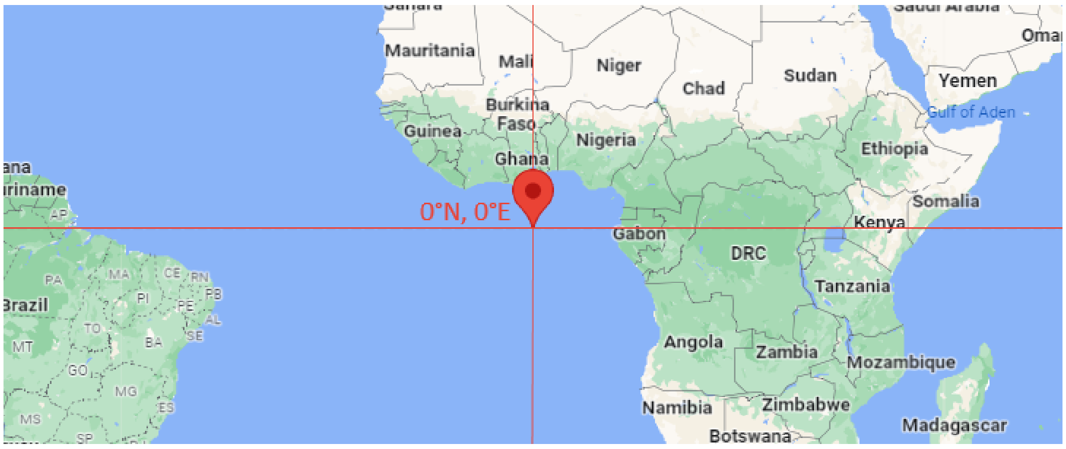


Figure 3. Point of intersection between the equator (horizontal) and prime meridian (vertical).

The ECEF system is used in conjunction with a geodetic datum such as the World Geodetic System (WGS84), which provides the reference ellipsoid for the Earth. As mentioned, ECEF coordinates are 3D Cartesian coordinates with the center of the Earth as the origin. While ECEF coordinates are useful for locating objects around the Earth such as satellites, they are not especially suited for use on the surface. This is primarily due to how ECEF coordinates are defined, i.e., using an X-, Y-, and Z-coordinate in relation to the center of the Earth. This means that when defining a point on the Earth's surface using ECEF, the reference ellipsoid always has to be taken into account, as to calculate the value for the individual axes. While a computer is easily able to handle such a problem, it can be counterintuitive for a human, especially if the objective is to define a point on a 2D surface. To solve this issue, coordinates measured in the ECEF system are transformed into *ellipsoid coordinates*. In the rest of this thesis, whenever WGS84 coordinates are mentioned, it is implied that they are formatted as ellipsoid coordinates.

2.4.2 Ellipsoidal coordinate system

The Ellipsoidal Coordinate System defines coordinates by *latitude*, *longitude*, and *altitude* over a reference ellipsoid. Latitude is the angle between two lines originating at the center of the Earth and extending out towards two points on the meridian plane, where one of the points exists on the equator. Longitude is measured in the same manner as latitude, but with two points in the equatorial plane, where one point exists on the prime meridian. Latitude is positive towards the north, while longitude is positive towards the east. The altitude is the distance from a point on the reference ellipsoid to a point on the surface of the Earth. The line that these two points form should be perpendicular to the surface of the ellipsoid[7]. Although an ellipsoid coordinate is defined using three axes, only the first two, i.e., latitude and longitude, are needed to reference a point on the surface of the Earth. This is the format that most people are familiar with, as it is widely used in consumer-oriented products such as "Google Maps".

2.4.3 Universal Transverse Mercator coordinate system

Up until now the focus has been on clarifying the systems related to geographic coordinate systems. Universal Transverse Mercator (UTM)[2] is a type of projected coordinate system that splits the Earth into 60 different zones, each covering six degrees on the equatorial plane. These zones are labeled with a number, projected onto a plane using a horizontal cylindrical projection, and subsequently stitched together. The map is then segmented into a grid, with the zone number (1-60) signifying the X-axis and a letter (A-Z) signifying the Y-axis. Zones 1-30 are east of the prime meridian while zones 31-60 are to the west. Letters A-M are south of the equator while letters N-Z are to the north. For instance, the cell 32U contains most of Germany, while the U.S. state of Florida is contained in cell 17R. Finding a coordinate inside a grid cell is as simple as locating its X- and Y-coordinate, which are defined in meters. As each zone's meridian runs through its center, an offset of 500 000 meters is used to avoid negative X-coordinates inside a grid cell. This is called a "false easting"[2].

2.4.4 Tile Map Service

Tile Map Service (TMS) is a method of subdividing a projection of the Earth into different levels of detail. Level zero, the root level, contains a square tile of the entire Earth, while level one splits the tile at the center of both the X- and Y-axis, dividing the tile into four new tiles. While TMS is not useful in itself as a precise coordinate system, it is used extensively in digital maps, such as Google Maps. When zooming in on a digital map, the images are replaced dynamically to display the highest detail map necessary for that specific level of zoom. It also doubles as a way to reduce the number of images the user needs to load to view the map, as only the tiles that are visible on the screen are necessary.

The indexing method for these tiles is quite simple. It uses a "level" in addition to X- and Y-coordinates. The number of tiles increases with the level, as lower-level tiles are split into four smaller ones for each level increase. When a tile is split, the X- and Y-coordinates of the new tiles become the same as their parent tile, multiplied by 2. The new tiles not overlapping the origin of the parent tile gain an offset of one in either or both the X- and Y-axis. In the official TMS implementation, the Y-axis increases upwards, while the version of TMS that Google uses increases downwards.

The method by which tiles are indexed makes for a simple file system structure for storing tiles. At the root there are directories for all desired levels, labeled according to their depth. Inside the level directories there are directories for all necessary columns, i.e., the X-coordinates. Finally, inside the column directories there are files representing the tiles, labeled according to the row, i.e., the Y-coordinate. For instance, accessing the map tile image for $X = 17$ and $Y = 21$ at level 5 involves locating the level directory labeled "5", then the column directory labeled "17", and finally locating the file labeled according to the row, "21.png". This simplicity enables developers to integrate TMS into their applications quickly and in an error-proof way.

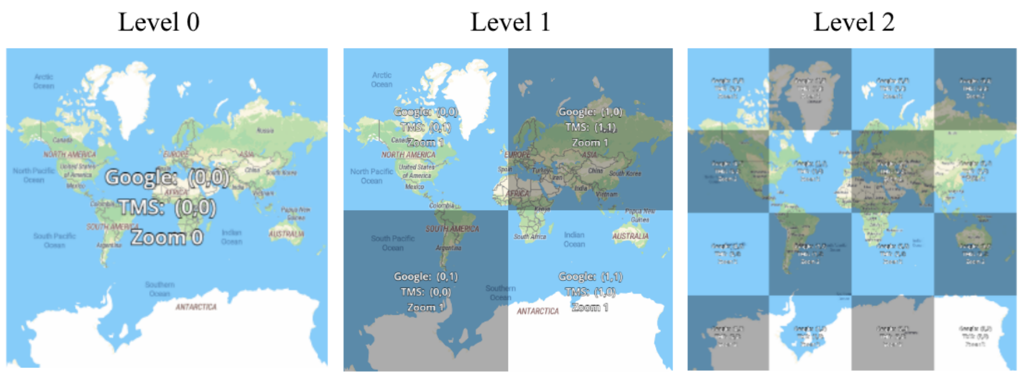


Figure 4. Subdivision of TMS tiles with the increase of level.

Chapter 3

Quantized Mesh

3.1 Overview

The QuantizedMesh format, developed by Cesium GS, has become somewhat of an unofficial standard for encoding 3D geographic data with multiple levels of detail (LOD). The reason why this is the case is primarily due to the fact that the Cesium platform has seen a great deal of success over the years as it was able to provide “the world’s most accurate, performant, and time-dynamic virtual globe”[8]. A QuantizedMesh file uses the extension “.terrain” and represents a square tile of geometry. Although the tile’s header specifies the location of the tile, the data it contains is not bound to a specific location or place. It is primarily a collection of vertices and indices defining a 3D mesh (Fig. 5). A QuantizedMesh file is composed of little-endian binary data, structured into multiple different sections in the following order:

1. Header
2. Vertex data
3. Index data
4. Edge index data
5. Extensions (optional)

As the name of the format suggests, the data in a tile is *quantized*. Quantization is a form of lossy compression where a set of values is segmented into subsets where those values are represented by a single value[14]. Quantizing data is preferable in the case that a small file size is more important than maintaining a high level of detail in the data, as values can be encoded using fewer bytes. Keeping the file size low achieves more data throughput, which improves the feasibility of streaming tiles over the internet. This approach is seen in many aspects of the design of this format, where various compression techniques are used to further reduce size at the cost of some processing power during tile creation and decompression.

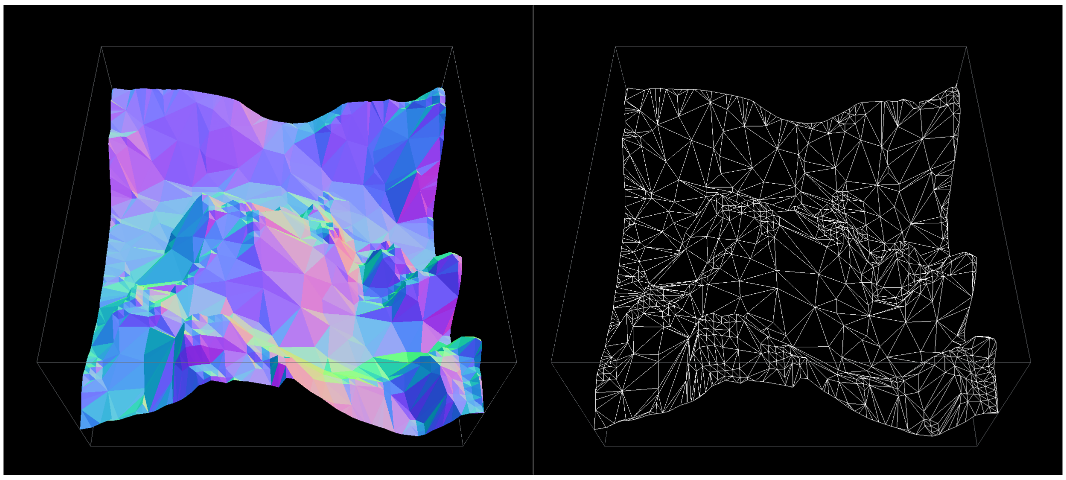


Figure 5. Quantized Mesh tile displayed in a 3D viewer.
Left - tile with colored planes. Right - tile as wire mesh.

3.2 Header

The header of a tile contains data that provides a viewer with the information needed to calculate when and how to display the mesh. The most vital parameters of the header are the minimum and maximum height of the vertices described in meters, as it allows the viewer to position tiles at the correct height in relation to each other as well as to properly scale the actual height values of the tile.

In order to figure out whether a tile should be displayed or not, a bounding sphere as well as a horizon occlusion point present in the header of the tile can be utilized. The bounding sphere is the smallest sphere that contains the entirety of a mesh in a tile. If this sphere exists anywhere outside of the view frustum (a 3D volume that contains everything visible on the screen[19]), then it means that the tile is outside of screen space and should not be rendered. The horizon occlusion point, which is expressed as an ellipsoid-scaled coordinate, is used to calculate whether the tile is currently beyond the horizon. If this is indeed the case, then the tile will not be rendered. Utilizing this type of precomputed information means that, when rendering, it is not required to perform back-face culling for tiles beyond the horizon where not a single triangle is visible, thus improving performance. The actual position of the tile is given as its center point expressed as an Earth-centered, Earth-fixed (ECEF) coordinate.

3.3 Vertex Data

Directly after the header is the vertex data section. A vertex is a point where two or more lines or edges meet[18]. In this thesis, when referring to a vertex, it is implied that it is part of a 3D mesh. The data in this section is split into three arrays: U-coordinates (X-axis), V-coordinates (Y-axis), and heights (Z-axis). The arrays are all of the same length, as specified by an unsigned integer at the start of the vertex data section (Table 1).

Vertex Data		
Field	**Type**	**Size**
vertexCount	Unsigned Integer	4 bytes
u	Array of Unsigned Short	2 bytes * vertexCount
v	Array of Unsigned Short	2 bytes * vertexCount
height	Array of Unsigned Short	2 bytes * vertexCount
Theoretical section length: 4 + 3 (2 * X) bytes		

Table 1. Vertex Data section

Together, these arrays contain the X-, Y-, and Z-coordinates of a set of vertices. These vertices represent WGS84 coordinates with the axes normalized to the bounds of the tile and scaled by 32767, effectively placing all values in the range of 0 and 32767. By normalizing the coordinate values from 8-byte doubles to 2-byte unsigned shorts in this range, the vertex data size can be reduced by a significant amount while still maintaining an acceptable level of detail for convincing terrain visualization. Using the formula:

$$\frac{max - min}{32767}$$

where *max* is the greatest and *min* is the smallest value present for some axis in this particular set of vertices, it is quite clear how to estimate the degree to which values can be represented. E. g. for a tile with a height range ($max - min$) of 350 meters, each meter could be expressed with a precision of approx. one centimeter. This process of losing detail but reducing tile size is what was described in section 3.1 as *quantization*. Finally, when encoding the vertex data, the values are all represented as the difference from the previous value which is then zigzag encoded within the tile.

3.4 Index Data

The next section in the QuantizedMesh file is the index data, which describes how the vertices are connected to one another. In the vertex data section, the vertices were stored in arrays. The indices to those vertices are now used in this section to refer to a specific vertex, which eliminates the need to declare the same vertex multiple times. This is a very common method used to define meshes and is used by other widely used mesh formats such as Wavefront (.obj)[17]. This is also the preferred way to handle meshes in graphics APIs such as OpenGL[13], as it improves performance when rendering by allowing the GPU to better utilize its cache.

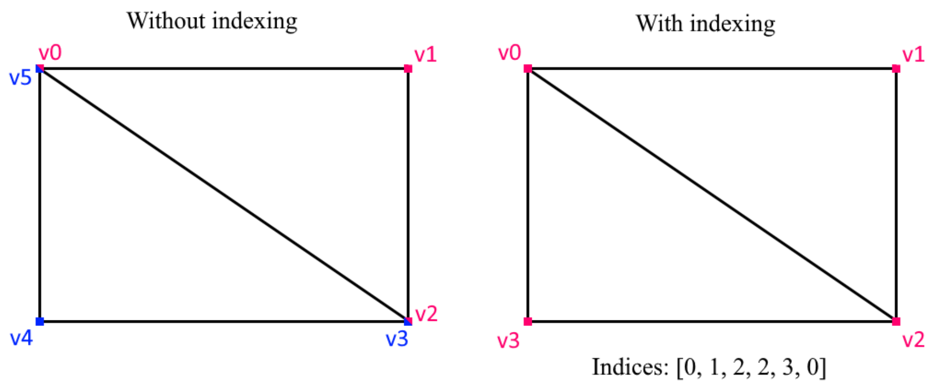


Figure 6. The number of vertices needed without and with indexing

Each triplet of indices corresponds to one triangle in the mesh. These triplets are stored in an array one after another (Fig. 6). Therefore, the length of this array is the number of triangles present in the mesh, multiplied by three. This section can be encoded in two different ways: storing the indices as unsigned shorts (2-byte), or as standard 4-byte integers. While the default is to use unsigned shorts, in some cases, when the mesh contains more than 65535 vertices, using only two bytes to store each index is no longer possible as it exceeds the maximum value of a 2-byte integer. Encoding more than 65535 vertices will therefore require the use of 4-byte integers as indices. A mesh exceeding 65535 vertices is therefore not advisable, as it effectively doubles the size of the index data section (Table 2). Thankfully, the vast majority of tiles do not require such a large number of vertices to clearly visualize their geometry. As a reference, there are approximately 1000 vertices in any given tile produced during the course of this project.

Index Data 16		
Field	**Type**	**Size**
triangleCount	Unsigned Integer	4 bytes
indices	Array of Unsigned Short	2 bytes * 3 * triangleCount
Theoretical section length: 4 + (2 * 3 * X) bytes		
Index Data 32		
Field	**Type**	**Size**
triangleCount	Unsigned Integer	4 bytes
indices	Array of Unsigned Integer	4 bytes * 3 * triangleCount
Theoretical section length: 4 + (4 * 3 * X) bytes		

Table 2. Index Data section (16- and 32-bit versions)

3.5 Edge Indices

This section contains the indices of all the vertices that are located on the edge of the tile. The structure is similar to the previous section of index data, as the values can be encoded using either 2-byte or 4-byte indices (Table 3). However, the reason why it is desired to keep track of this kind of data is quite different compared to the previous section. Knowing which vertices exist on the edge is mainly important for visual fidelity but can also have an impact on various geometric calculations due to surface inconsistencies. Therefore, it is vital for tiles to have their edges line up correctly in order for them to form a continuous surface without gaps. Edge vertices should optimally overlap between neighboring tiles. The reason why these indices are maintained separately from other index data is due to various different use cases, e.g., when sampling tiles, instead of having to perform expensive checks on each vertex to detect if it exists on the edge during runtime, the information is already available in a separate precomputed section of the file. The trade-off is that file sizes are slightly larger than otherwise required. In addition to being useful during tile processing, edge indices could also be used to easily stitch multiple tiles into one connected mesh, which can be useful in cases when tiles are not optimal.

Edge Indices 16		
Field	**Type**	**Size**
westVertexCount	Unsigned Integer	4 bytes
westIndices	Array of Unsigned Short	2 bytes * westVertexCount
southVertexCount	Unsigned Integer	4 bytes
southIndices	Array of Unsigned Short	2 bytes * southVertexCount
eastVertexCount	Unsigned Integer	4 bytes
eastIndices	Array of Unsigned Short	2 bytes * eastVertexCount
northVertexCount	Unsigned Integer	4 bytes
northIndices	Array of Unsigned Short	2 bytes * northVertexCount
Theoretical section length: 16 + (2 * (A + B + C + D)) bytes		
Edge Indices 32		
Field	**Type**	**Size**
westVertexCount	Unsigned Integer	4 bytes
westIndices	Array of Unsigned Integer	4 bytes * westVertexCount
southVertexCount	Unsigned Integer	4 bytes
southIndices	Array of Unsigned Integer	4 bytes * southVertexCount
eastVertexCount	Unsigned Integer	4 bytes
eastIndices	Array of Unsigned Integer	4 bytes * eastVertexCount
northVertexCount	Unsigned Integer	4 bytes
northIndices	Array of Unsigned Integer	4 bytes * northVertexCount
Theoretical section length: 16 + (4 * (A + B + C + D)) bytes		

Table 3. Edge Indices section (16- and 32-bit versions)

3.6 Extensions

The last section of a QuantizedMesh tile is the optional extensions. Every type of valid extension has a 1-byte integer identifier to express what kind of data it contains. The actual structure of the extension data can be of any type, which is why the extension header only contains an integer expressing the length of the section of data that is to come, expressed in bytes (Table 4).

Extension Header		
Field	**Type**	**Size**
extensionId	Unsigned Char	1 byte
extensionLength	Unsigned Integer	4 bytes
Theoretical section length: 5 bytes		

Table 4. Extension Header section

The actual extension data could in theory be virtually anything. The only requirement is that the identifier (extensionId) does not collide with any other type of extension. There are however a few officially supported extensions available. One example is the metadata type (Table 5). The data in this extension is formatted as a JSON string and could essentially contain any type of data. However, its main use case would be to describe the geographic properties of a tile[15]. Using this extension, a developer could, for instance, implement functionality to select an area or terrain and be presented with what kind of vegetation or bedrock is present there. Any type of data that needs to remain human-readable would be suitable to store in this extension.

Metadata (extensionId = 4)		
Field	**Type**	**Size**
jsonLength	Unsigned Integer	4 bytes
json	Char	1 byte * jsonLength
Theoretical section length: 4 + X bytes		

Table 5. Metadata section

Chapter 4

Project Introduction

4.1 Project overview

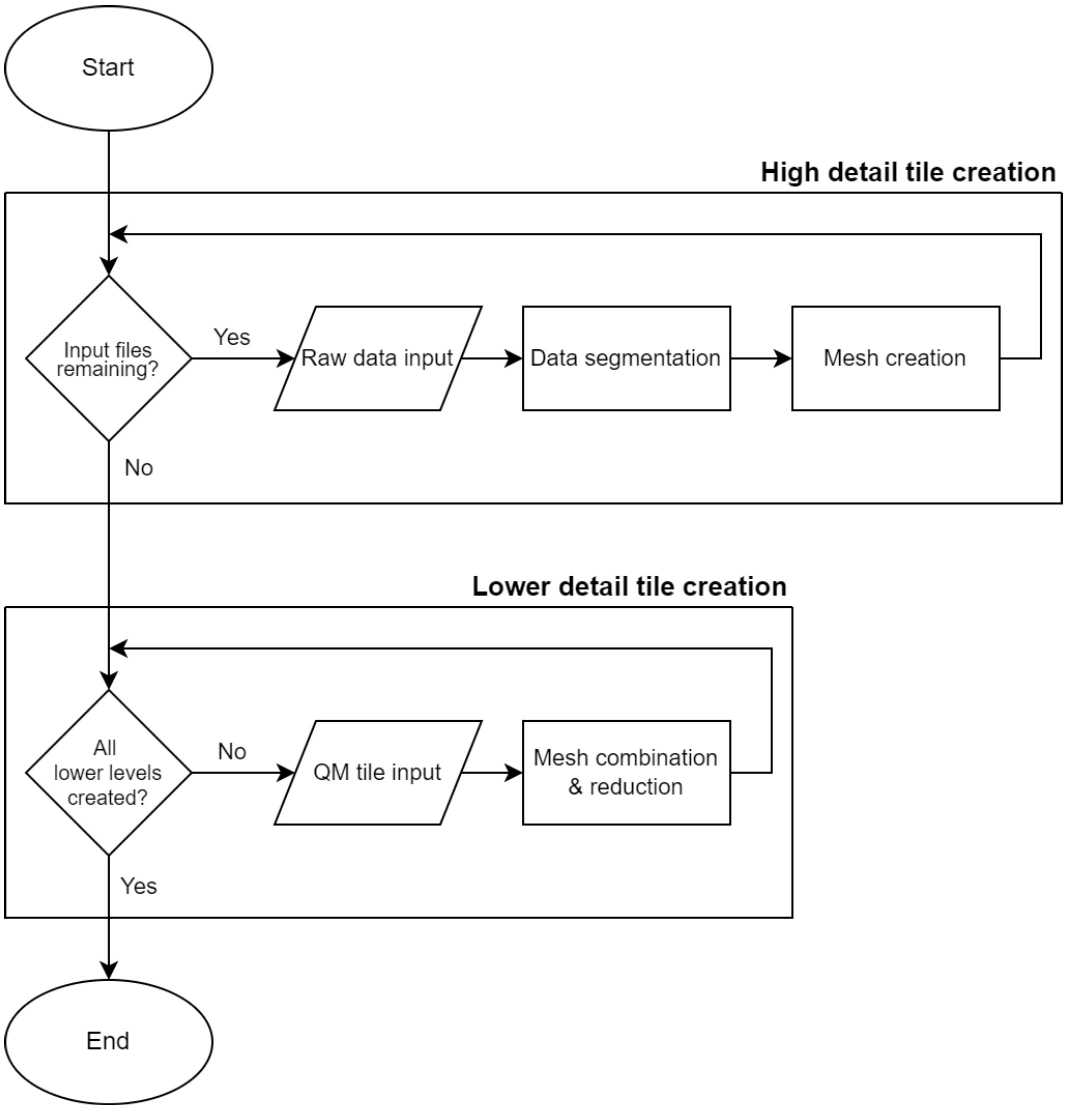
The project for this thesis is to build a software tool that transforms large amounts of geographical height data into multiple levels of QuantizedMesh tiles of varying levels of detail. The main area of interest that is to be transformed is the surface of Finland. The file format of the Finnish height data is ASCII grid, while the contents are ETRS-TM35FIN coordinates (UTM coordinates in zone 35), and N2000 height values (relative to mean sea level)[16]. However, despite the focus on publicly available Finnish height data, the application should also be able to easily be adapted to handle other kinds of formats used in different contexts. The mesh itself should optimally be a *Triangulated Irregular Network* (TIN), which is a mesh where only vertices that have a large impact on the overall shape of the mesh are retained. With a TIN mesh, the file size is smaller while the visual clarity remains roughly the same.

The reason for developing this project was to have a consistent and flexible way of creating QuantizedMesh tiles for use in other applications. For instance, whenever there is a need to visualize geometry or utilize line of sight, having this type of data available aids in quickly and intuitively adding the desired functionality.

4.2 Design

As for the design of the application, the overall process is split into two major segments: creation of the highest level of detail tiles, and the creation of all other requested lower-level tiles (Fig. 7). The reason for this decision was a few different factors. First, this method requires programs to read and process the input data only once, as it is only necessary when creating the initial high detail tiles. This is due to the fact that when creating the lower levels, the high detail tiles that were just produced can be used to sample data which has already been processed. The high detail tiles are combined and reduced to create tiles that cover a larger area, yet contain less detail. Secondly, handling each level separately instead of processing all levels for each input file at once reduces the number of tile mergings necessary. Instead of a need to merge tiles for every level, it is only needed during the first (i.e., highest) level.

Another important decision was to decouple the input handling section from the tile creation pipeline and use a common type of input for the data processing section. This was to allow for easier future implementations of other input data formats and is generally a good way to structure an application. As QuantizedMesh tiles are designed to use WGS84 coordinates, this was a natural choice as the common input type.



Start
High detail tile creation
Input files remaining?
Yes
Raw data input
Data segmentation
Mesh creation
No
Lower detail tile creation
All lower levels created?
No
QM tile input
Mesh combination & reduction
Yes
End

Figure 7. High level diagram of the entire application pipeline.

4.2.1 Creation of tiles

As mentioned, the objective is to transform height data from large geographical areas into QuantizedMesh tiles. The tiles are to be structured and indexed using TMS, which features its own grid system, i.e., the QuantizedMesh tiles that are produced should coincide with the TMS grid cells. However, input data are not guaranteed to align with the TMS grid. If the data remains axis-aligned, only minor steps need to be taken to adhere to the TMS grid. In those cases, data can be loaded and processed in chunks that themselves are perfectly aligned with the grid. What causes the majority of issues is whenever the input data is tilted or distorted and i no longer axis-aligned with the TMS grid, such as when converting coordinates from one system to another (Fig. 8). The data now has to be treated as a simple set of coordinates, which eliminates numerous assumptions that could otherwise be made regarding the data.

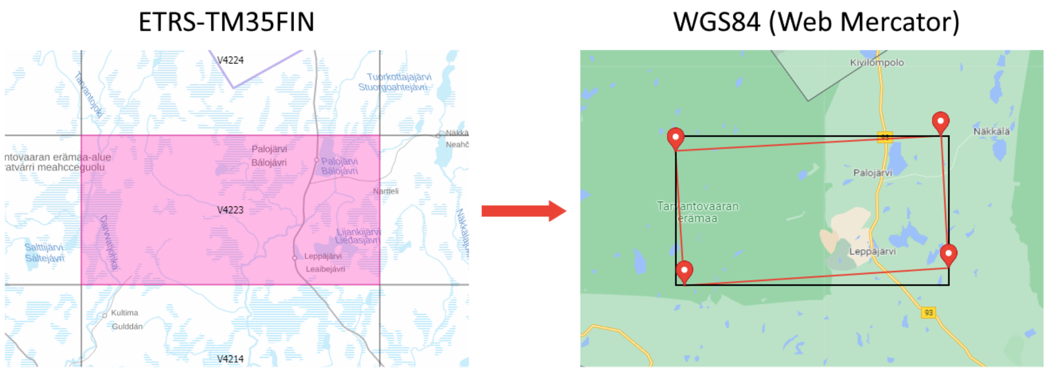


Figure 8. Corner coordinates of ETRS-TM35FIN rectangle in WGS84 (Web Mercator). The red rectangle on the right represents the left rectangle transformed into WGS84. This showcases how the rectangle is no longer axis-aligned after transformation.

Dealing with non-aligned data can be rather difficult. If the input files are naively loaded and processed one by one, it will result in many unfinished QuantizedMesh tiles. This is due to the fact that the input data only partly covers the TMS tile. Furthermore, whenever processing input data of areas that are adjacent to previously processed areas, the unfinished tiles on the edges between the regions of input data will be overwritten. One method to mitigate this problem is to load all input data files at once and process them as one large set of data. However, when processing large regions such as entire countries, the amount of memory needed to load all files at once is simply too large for most workstations (Table 6). Therefore, the solution that was chosen in the context of this project was to load the input files one by one but merge the data of the new and the existing tile into one whenever identified instead of overwriting the tile. This means that whenever a tile is about to be written to disk, a check is made to see if the tile already exists. If the tile exists, it is loaded and merged with the newly created tile before being written to disk. This way, no matter how many files need to be processed, the memory overhead will be roughly the same, at the cost of some performance. As for creating the lower level tiles, the process is quite straightforward. Using higher detail tiles as a source, the meshes are combined and subsequently reduced. This results in tiles covering the same area as the source tiles, but containing much fewer vertices, as that kind of detail is not as necessary when visualizing large areas. This process is repeated for every lower level until level 0 or for as long as the user has defined.

Data point density	**Area per input file**	**Max loaded size**	**Finland memory requirement**
10m	288 km²	57.6 MB	67.7 GB
2m	36 km²	180 MB	1.69 TB

Table 6. Table highlighting the amount of memory needed to keep data in memory. The area of Finland is approx. 338 440 km²

4.3 Tools

The decision of what tools to use for this project was quite clear from the beginning. The project was developed for Navielektro Ky and, as such, needed to fit their environment, which heavily utilizes Java. Beside that, some tools were found during the development process that proved to be very useful. Thus, the tools chosen for the development of this project were the following:

- Java
- Maven
- IntelliJ IDEA
- Quantized-mesh-viewer
- Maptiler.com
- Proj4j by locationtech
- Poly2tri

Maptiler.com and quantized-mesh-viewer were some tools found during development that ended up becoming essential to the overall development process. Maptiler.com provides a digital map that properly displays TMS tiles, their coordinates, and other useful information such as the bounds of the tile in WGS84 coordinates. This tool was vital when implementing the TMS system into the project, and to verify that the implementation was able to accurately index tiles. The tool also proved to be useful when there was any need to find the coordinates of specific tiles during debugging.

Quantized-mesh-viewer is an open-source project on GitHub that includes a QuantizedMesh implementation for the CesiumJS JavaScript library, as well as a QuantizedMesh tile debugger. The debugger can be used to inspect individual tiles on a deeper level. It includes features such as visualization of a wire mesh, triangle planes, vertices, edge vertices, and more. This tool was vital when implementing the mesh creation component of the application in order to see if tiles are displayed as intended.

Chapter 5

Project Implementation

5.1 QuantizedMesh I/O

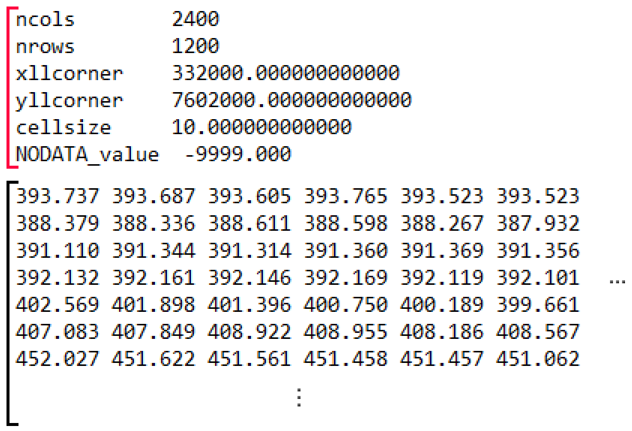
The first part of the project entailed implementing an encoder and a decoder for the QuantizedMesh format in Java. This would preferably have been avoided, but as there were not any readily available open-source implementations of the QuantizedMesh format for Java, there were unfortunately no other options. However, implementing the encoder and decoder proved to be especially useful in gaining a better understanding of the format right from the onset of the project. The experience gained during the format's implementation informed many of the decisions that were made during development. Aside from the experience, another positive aspect of implementing the format is the direct ownership of the code.

A slightly troublesome detail of the implementation of the format was the choice of programming language. The QuantizedMesh format contains plenty of unsigned variables which, unfortunately, Java does not support. However, despite this being the case, it is still possible to work with unsigned values in Java, but they must be stored in a data type at minimum one size larger. For instance, an unsigned short (2-byte) must be stored in a regular integer (4-byte) so as not to lose any information. Luckily, Java does provide some functionality to perform these conversions, but it remains far from ideal. While this does increase the memory usage of the program, not all benefits of using unsigned data types are lost. QuantizedMesh files stored on disk or transferred through the internet still benefit from being encoded using smaller unsigned data types as it reduces the size of each file to some degree.

The decoder ended up being implemented before the encoder. The motivation behind this decision is that this was the starting point of the development process, and at the time there was no way to produce any proper data. Therefore, by implementing the decoder first, it could be used to load sample ".terrain" files that could then be used as testing data when implementing the encoder. When implementing the decoder, multiple open-source QuantizedMesh decoders written in other languages were used as references to verify the validity of the decoded results. As for implementing the encoder, it became necessary to build a small tool to identify the differences between the original sample ".terrain" file and the newly encoded file. The tool would report the byte position at any instance where the two files did not perfectly match. By utilizing this tool, it became much easier to identify bugs during development.

5.2 Input data

The main format of the input data used in the application is ASCII grid, which is a non-proprietary version of the Esri grid format. As the name implies, the format is grid based and primarily used for storing elevation data. All cells in the grid are of equal size and are able to store one value. The format is entirely text-based and does not use any special method of encoding. As for the contents of the format, it contains a header, which covers the first six lines of the file, and a segment of raster values for each cell, delimited by single space characters (Fig. 9). The header contains metadata for the file. It includes parameters for the size of the grid, measured in columns and rows, and the size of each grid cell, defined in meters. Calculating the coordinates of the cells (bottom left corner of each cell) is quite straight-forward as well. The header contains parameters for the X- and Y-origin coordinates, and by knowing the size of each cell, take the x and y position of the cell in the grid, multiply them by the cell size, and add the results to the X- and Y-origin. Finally, the header contains a value for signifying that a cell has no data. If any cell contains this value, the cell is empty or contains some null data.



```
ncols         2400
nrows         1200
xllcorner     332000.000000000000
yllcorner     7602000.000000000000
cellsize      10.000000000000
NODATA_value  -9999.000
393.737 393.687 393.605 393.765 393.523 393.523
388.379 388.336 388.611 388.598 388.267 387.932
391.110 391.344 391.314 391.360 391.369 391.356
392.132 392.161 392.146 392.169 392.119 392.101  …
402.569 401.898 401.396 400.750 400.189 399.661
407.083 407.849 408.922 408.955 408.186 408.567
452.027 451.622 451.561 451.458 451.457 451.062
                          ⋮
```

Figure 9. ASCII grid format.
The red segment are the six lines representing the header.
The black segment is the grid cell values.

Parsing the data is fairly straight forward. As the header of the file always follows the same format, it can be assumed that the first six lines always contain the header, which can easily be parsed by removing white spaces on each line and splitting them accordingly to extract the required values. As for the grid of height values, the section is oriented as it would be when overlaid on a map, with the origin in the bottom left corner and the X- and Y-axis extending to the right and upwards. Parsing this section entails traversing the grid one line (i.e., row) at a time and extracting the values from the line by splitting at every space character. After extracting the height values, the coordinate for each cell is properly offset as explained in the previous segment. In this case, the coordinate data that these values represent is formatted to the ETRS-TM35FIN system. This is the official coordinate system used for anything regarding land maps localized to the country of Finland. The coordinate system uses UTM coordinates, which have to be converted into latitude/longitude coordinates in order to conform to the specifications of the QuantizedMesh format.

When performing the conversion, the horizontal datum used for the output coordinates have to be taken into account. In this case, the desired output coordinates are formatted according to WGS84. In practice, UTM coordinates were converted into WGS84 by using the Proj4j library. However, this only converts the X- and Y-axis of the coordinate. Additionally, the vertical datum also has to be taken into account, or it will lead to the QuantizedMesh tiles containing incorrect height data. This process is a bit more complicated than the horizontal conversion, as a valid geoid is needed to correctly offset the height. The height values supplied in the input data are measured as meters above the mean sea level around Finland. However, as WGS84 coordinates specify height values over the reference ellipsoid, these values have to be offset by the height of the geoid at that point. However, producing intuitively useful data representing the geoid is easier said than done. Although geoids are distributed to the public by various government agencies, they are usually not supplied in any useful form. Thankfully, the NGA (National Geospatial-Intelligence Agency), a US government agency, provides code to produce a .csv file from a reference geoid. By utilizing this data as a source, it is trivial to create an interface to sample the geoid height at specific lat/long coordinates. The height values sampled from the geoid are then used to offset the height over mean sea level to produce the proper ellipsoid height of the coordinate.

5.3 Data segmentation

5.3.1 First look

When segmenting the input coordinates into their respective locations, there were a few different ideas that needed to be explored in order to have the correct spread of data available during the mesh creation process. The main idea is to place coordinates into the correct TMS tile according to the location given by the coordinate. As previously mentioned, there is no guarantee that the transformed input data is aligned with the TMS grid. This unfortunately means that the input data, which is stored in standard arrays, cannot be split into segments using array indices as the collection essentially functions as a set of points. Instead, to guarantee that the correct coordinate is placed into the correct TMS tile, every coordinate needs to be checked individually to verify its location. Processing the data in this manner results in a set of TMS tiles containing coordinates that exist within the perimeters of the tile.

5.3.2 The problem with gaps

While the above method accomplishes the goal of segmenting coordinates into their respective tiles, it fails to address a major issue, i.e., enabling the preservation of the edges of the tiles. Preserving the edges of tiles is important for properly representing the geometry and plays a large part in the overall complexity of the project. At this instance, because the input data is not necessarily aligned with the TMS grid, coordinates rarely exist perfectly on the edges of the tiles. This results in large gaps between meshes, which compromises the quality of the final QuantizedMesh tiles (Fig. 11).

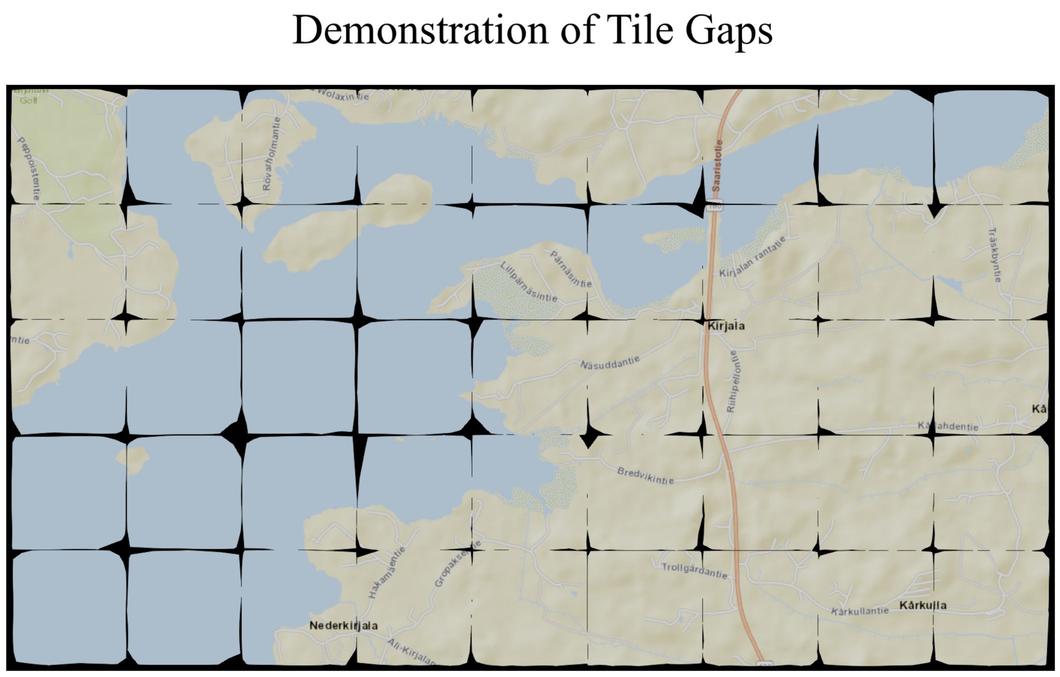


Figure 11. Gaps between QuantizedMesh tiles without preserved edges. Tiles displayed in Cesium.

Currently, the segmented collections of coordinates are completely independent of one another. This is positive as it allows for easily processing tiles in parallel without any major complications. However, this also means that during the processing of tiles in the mesh creation stage, there is no way of accessing neighboring tiles to "stitch" them together and remove the gaps. As this is a two-part issue, starting with how data is segmented and concluding with how that data is processed, it was necessary to analyze what kind of information is required to stitch the tiles together. In most cases, coordinates do not exist perfectly on the edges of tiles, but that data is still required to properly construct the mesh. Therefore, if the coordinates right outside the tile's perimeters are known in addition to those inside the bounds, the points existing perfectly on the edges can be interpolated. This means that the current method of segmenting the coordinates must be expanded upon to also check where in a given tile a point is located, and if that point should be copied to any of the neighboring tiles.

5.3.3 Algorithm extension

To implement the new coordinate segmentation requirements, two additional bounds were added to every tile: the *inclusion bounds* and the *exclusion bounds*. These two bounding boxes are defined in relation to the previously established TMS tile bounds by either adding or subtracting a small offset to its origin and dimensions. The inclusion bounds define an area that surrounds the actual area of the tile, while the exclusion bounds exist within the tile (Fig. 12).

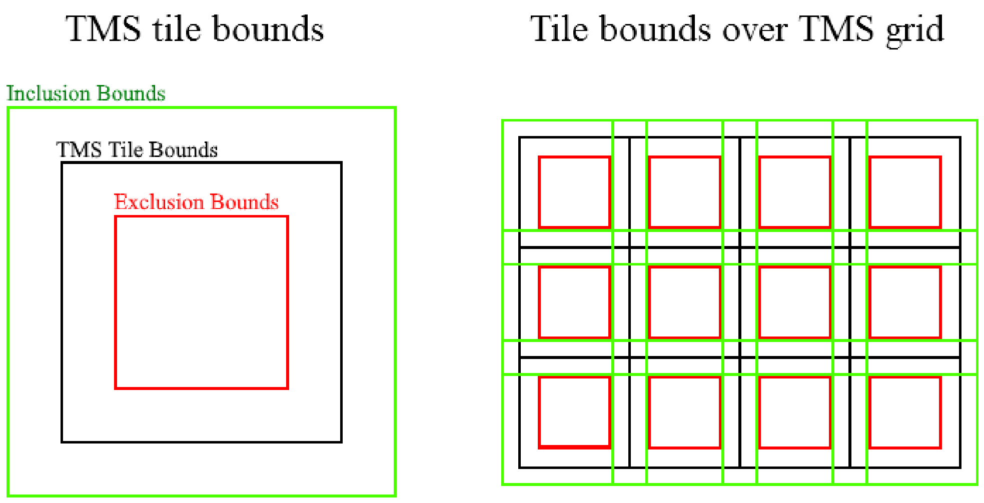


Figure 12. The additionally defined TMS tile bounds and their coverage.

The purpose of these new bounding boxes is to be able to categorize coordinates in relation to a tile. A coordinate is always added to at least one tile. When that coordinate exists within the exclusion bounds of a tile, then the coordinate only belongs to that specific tile and should not be evaluated further. When a coordinate exists inside the tile bounds but outside the exclusion bounds, the coordinate belongs to that specific tile, but should be further evaluated to check if it also exists inside the inclusion bounds of any neighboring tiles. If it does, then the coordinate should be added to those tiles as well (Alg. 1). While the inclusion bounds are required for finding coordinates around the perimeter of tiles, the exclusion bounds are optional. Its actual purpose is to avoid performing expensive checks on neighboring tiles for every coordinate. Considering that most coordinates are located inside the exclusion bounds, it results in quite a performance gain, reducing the segmentation time by approx. 50%.

```
function PLACECOORDINATE(Coordinate, Zoom)
    $Tile \leftarrow GetTileForCoordinate(Coordinate, Zoom)$
    Add Coordinate to Tile.Coordinates
    if Coordinate not within Tile.ExclusionBounds then
        for each NeighborTile do
            if Coordinate within NeighborTile.InclusionBounds then
                Add Coordinate to NeighborTile.EdgeCoordinates
            end if
        end for
    end if
end function
```

Algorithm 1. How each coordinate is placed into the correct tiles.

Now that all coordinates have been segmented into the correct tiles, there is but one final piece of information missing: the minimum and maximum height of the coordinates present in each tile. Thankfully, this information can easily be calculated while performing the coordinate segmentation. Whenever a coordinate is regarded as belonging to a tile and subsequently added, its height is compared to the current minimum and maximum height recorded in the tile. If the height of the coordinate is less than that of the current minimum of the tile, the value is replaced. Similarly, if the height is greater than the current maximum, the value is replaced. Once the data segmentation is finished, every relevant tile that is now ready to be processed contains the following information:

- Coordinates inside the tile
- Coordinates around the perimeter of the tile
- The minimum and maximum height of the coordinates present inside the tile
- The bounds of the tile (x, y, width, height), in WGS84 format

5.4 Mesh creation

Once all points have been segmented into their correct tile, the mesh creation process can begin. All collections of segmented points are completely independent of one another, which allows for easy parallelization of this section of the pipeline. Parallelizing the process can lead to great speedups depending on the number of cores available in the CPU. On a modern desktop CPU, featuring six cores and 12 threads, a speedup of a factor of around three was achieved. This speedup is quite significant, as the time to process very large areas, such as entire countries, can still be up to several hours.

Currently, the points are using latitude and longitude for the X- and Y-axis. Longitude values are defined as decimal values in the range of -180.0 to +180.0, while latitude covers the range -90.0 to 90.0. However, the Z-axis, which contains height values, has no range of values to adhere to. This means that variations in the Z-axis can be much larger than in the X- and Y-axis, which can potentially lead to inconsistencies during triangulation as all three axes are not scaled to one another. To mitigate this issue, the axes are normalized to the range [0,1] by utilizing the minimum and maximum values that were computed during the data segmentation process. As the normalization of these values has to be performed regardless, scheduling the process to occur before triangulation rather than after solves the issue and avoids any expensive scaling operations.

5.4.1 Triangulation

Triangulation is the process of calculating how points are connected to one another. The results of these calculations is what is used to create the index data section of QuantizedMesh tiles. Essentially, triangulation is the process of creating a mesh from a set of points/coordinates. While there are many different methods of triangulation, one commonly known algorithm is "Delaunay triangulation", named after Boris Delaunay, a Soviet mathematician who made important contributions to the topic from 1934 onward[1]. As standard mesh triangulation is a heavily researched topic with a great deal of available open-source software resources, there was no reason to implement the triangulation algorithm from scratch. Therefore, to avoid any potential bugs and speed up the overall development process, the Java version of the open-source library "Poly2Tri" was chosen as the triangulation library for this project. The library implements a triangulation algorithm based on the paper "Sweep-line algorithm for constrained Delaunay triangulation" by V. Domiter and

B. Zalik[3].

The normalized point data is handled by the library as a point set with no pre-existing connections or relations. After triangulation, the user can access information about the finished mesh, such as triangles present in the mesh, the points that make up those triangles, and the indices of those points. This data is later used as input for constructing the different segments of the QuantizedMesh tile.

5.4.2 Edge preservation

Preserving the edges of tiles is a major component of the mesh creation process. As mentioned in section 5.3.2, the issue with not properly preserving the edges of tiles is that it results in tiles not lining up in the correct manner and leaving large gaps between the tiles. This is not only visually jarring when viewing terrain tiles but can also cause inconsistencies when using the data for analysis. Therefore, eliminating the gaps between tiles is of utmost importance. As previously mentioned, the solution to this issue is a two-step process, starting with how data is prepared, and concluding with how it is processed. At this stage, every relevant tile contains a list of coordinates that exist inside the tile, as well as another list of coordinates around the perimeter of the tile. The reason for collecting coordinates existing outside the tile was to utilize them to interpolate any coordinates that exist precisely on the edge.

The idea is to triangulate all coordinates into a mesh that, in the X- and Y-axis, slightly extend past the bounds of the tile. The next step would be to find all triangles in the mesh that cross the border of the tile in any manner. The points that make up these triangles can then be used to define vectors originating inside the bounds of the tile and extending outwards. These vectors can then be utilized to find the intersection point on the border of the tile. Although the concept itself does not necessarily require anything extraordinary to be implemented, there are many pitfalls that need to be considered, especially the implementation of various geometric equations(Fig. 13).

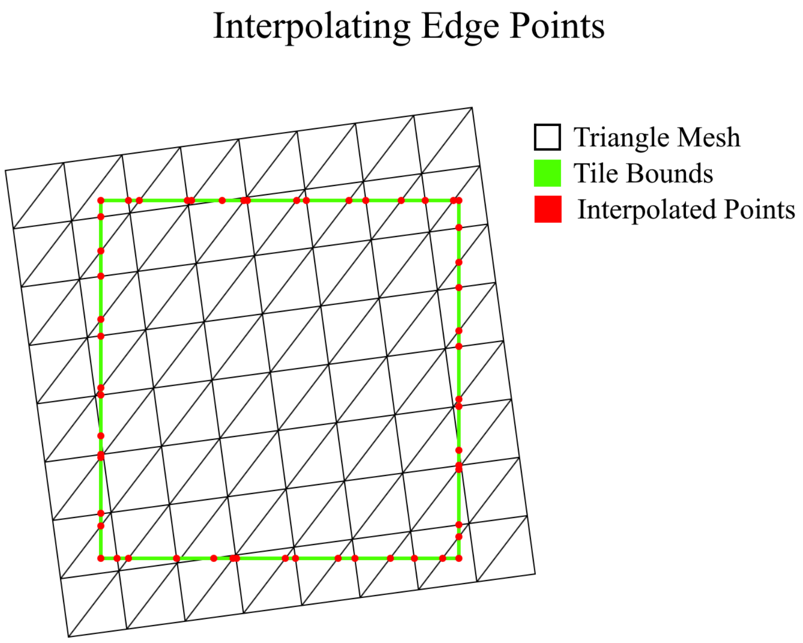


Figure 13. Interpolating points that appear on the edge of the tile. The triangle mesh is not aligned to the TMS grid.

Triangulating the coordinates results in a list of triangles present in the mesh. To find the triangles that cross the edge of the tile, the triangles are investigated one by one. The points that make up these triangles are checked to establish whether they are located inside or outside the tile perimeters. When a triangle contains some points on the inside and some on the outside, it means that the triangle crosses the perimeter. By keeping track of the inside and outside points, it is possible to identify all pairs of points that make up lines which are guaranteed to cross the border. These pairs of points can be utilized to define vectors that originate inside the bounds of the tile and extend outwards. The inside points serve as the origin of the vector while the outside points are used to set the direction of the vector. Conveniently, as all points are normalized to the bounds of the tile, it can be assumed that the axes of the tiles range from 0 to 1. This information is utilized to define the planes and normals of a box that surrounds the perimeters of the tile. Now all that is left is to calculate the intersection between the vectors and bounding box planes. The resulting point of intersection is a point that exists precisely on the edge of the tile. This process is repeated until all triangles have been investigated.

While this method works perfectly to interpolate the points that exist on the edges, it fails to find the points located at the corners. It is vital that the tile, when observed from above, is a perfect square. Without the corner points, there will be gaps between tiles where the corners meet. To solve this issue, whenever a triangle is confirmed to cross an edge, an additional check is made to see if any of the four corners of the tile exists within the triangle. The method that was chosen for solving this issue is quite elementary. First, the area of the triangle is calculated. Secondly, three new triangles are created by connecting the corners of the original triangle to the point of interest. If the sum of these three areas is equal to the area of the original triangle, it can be confidently stated that the point of interest exists inside the original triangle.

If, as a result of the corner check, a corner of the tile is confirmed to pass through the triangle, then an additional calculation needs to be performed to find the point of intersection. Fortunately, in this case, there is quite a bit of information that can be utilized to aid in finding the point. First, the triangle provided by the previous step is confirmed to exist on the corner. Secondly, the X- and Y-coordinates of each corner are known. By utilizing this information, the following equation can be used to find the Z-coordinate of a point on the triangular plane:

When the following parameters are known:

- *Points A, B, and C of a triangle confirmed to exist on the corner*
- *X- and Y-Coordinates of the corner/intersection point*

The following formula can be utilized:

$$\begin{aligned} z = A_z &+ \frac{(B_x - A_x)(C_z - A_z) - (C_x - A_x)(B_z - A_z)}{(B_x - A_x)(C_y - A_y) - (C_x - A_x)(B_y - A_y)}(y - A_y) \\ &- \frac{(B_y - A_y)(C_z - A_z) - (C_y - A_y)(B_z - A_z)}{(B_x - A_x)(C_y - A_y) - (C_x - A_x)(B_y - A_y)}(x - A_x) \end{aligned}$$

Once the point of intersection is known, the point is saved in the same manner as the other interpolated points, which in conjunction with one another, form the border of the tile and coincide with the edges of neighboring tiles, thus eliminating gaps.

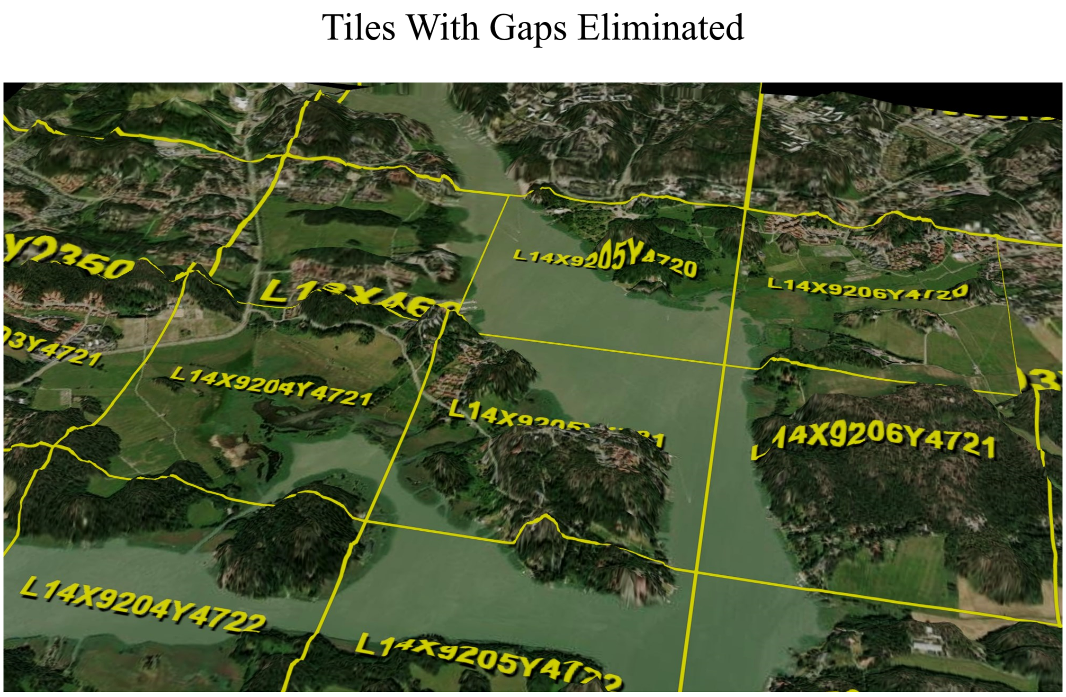


Figure 14. Interpolating points that appear on the edge of the tile.
Tiles displayed in Cesium (height exaggerated by factor of 10).

5.4.3 Mesh merging

As input files are handled one by one, there are multiple instances per file where the resulting QuantizedMesh tiles are on the edge of the area that the coordinates cover. This means that many tiles are only partially covered by points as the remaining points need to be sourced from another file. To complete a partial tile, multiple instances of that tile containing various parts of the final mesh need to be merged into one. In practice, whenever a mesh is about to be written, a flag is checked to see if the tile should be overwritten or merged. In the case that the tile should be merged, only a few steps need to be performed. First, the existing tile needs to be loaded, secondly, the tiles are merged into one, and finally, the tile is overwritten. As it is only possible to create one instance of a tile from any given input file, only at most two tiles will ever be merged at any point in time.

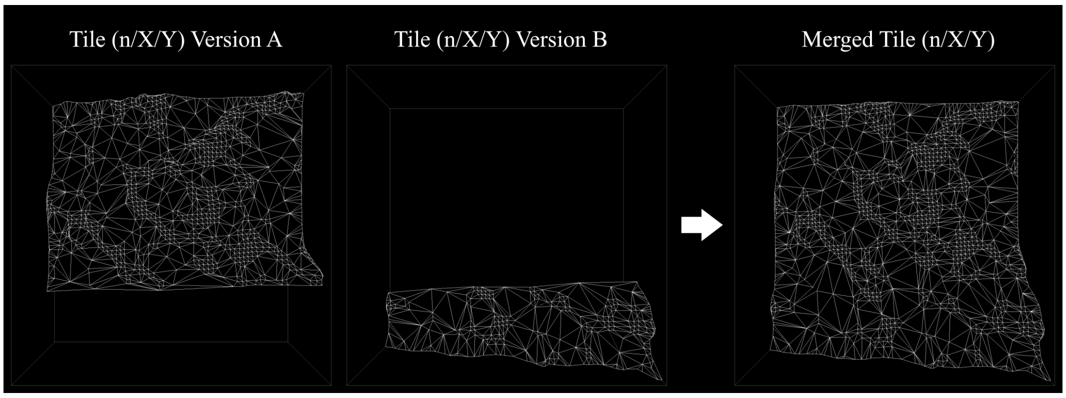


Figure 15. Interpolating points that appear on the edge of the tile.
Tiles displayed in debug viewer.

While this method does introduce a small performance overhead by loading and merging tiles, it allows for processing one input file at a time and using much less memory. The merging operation begins by loading and decoding the existing tile that is to be merged with the current tile instance. It then proceeds by denormalizing the height values of the vertices from both tiles back into their original values. The reason for performing this step is due to the fact that both tiles have been normalized according to the maximum and minimum height of their own set of vertices which most likely differ. If the vertices were combined into one set of points and triangulated, the final mesh would not properly reflect the actual terrain. For instance, in the case that one part is normalized according to the range of 15 to 30, while another part is normalized to the range of 10 to 100, the part of the mesh with a smaller range will have its height scaled incorrectly when merged with the other part. However, if all height values are denormalized back into their original form, they can then be renormalized using an updated height range and thus retain the shape of the terrain. Once the points have been denormalized, they are placed into a combined set. Additionally, as these tiles have already been processed before, there is no need to interpolate the points on the edges once again. Finally, the collections of points are processed by the mesh creation pipeline as per usual, without the interpolation stage activated. At this point, the two tiles have been merged into one which replaces the old partial instance of the tile.

5.4.4 Mesh sampling

Up until this point, everything that has been covered pertains to how the initial highest level of tiles is created, modified, and optimized. This covers the first segment of the design presented in the previous chapter. However, to complete the second and final segment, all other layers of tiles have to be created as well. While there are many different ways to achieve this, the method used in this project was based on relying on previously computed data. By sampling previously constructed tiles, it eliminates the need for converting coordinates, segmenting them into tiles, interpolating edges, and finally optimizing the mesh.

In the TMS system, for each increase in the level of zoom, each tile is subdivided into four smaller tiles. In the same manner, when decreasing the level of zoom, four smaller and higher detail tiles can be used as a source to construct one new tile. Therefore, by combining tiles that have already been constructed into larger tiles, it is possible to skip most of the work needed to create a brand-new tile. However, the detail that needs to be established, is how to know which larger, lower detail tiles encompass the tiles that have already been created. This issue was solved by utilizing the way that TMS tiles are saved on disk (Zoom/X/Y.terrain). By walking through all files of a specific level of zoom and checking their paths, the coordinates of all tiles are simple to extract. Each tile will have a path that looks somewhat like the following: ./tiles/15/18420/23305.terrain. From this path, the X- and Y-coordinates are extracted and used to calculate the X- and Y-coordinates of the encompassing tile by dividing the coordinate values by a factor of two, multiplied by the number of steps between the levels of zoom. By performing this check on all file paths for the level that is being sampled, a set of keys for the new tiles has been created. Now, each key can be used to know which source tiles should be sampled to construct that tile. Conveniently, as each source tile only belongs to one encompassing tile, the notion of parallelizing the computation becomes quite intriguing. This is due to the fact that there will naturally be no data overlap between tiles and no risk for any collisions anywhere while sampling.

When all the necessary source tiles that are needed to build the new tile are loaded, the sampling process begins. The data in all tiles is normalized to the range 0 to 32767. Thus, if the vertices of the source tiles are naively combined, the resulting tile will contain the data from all four tiles stacked on top of each other. To avoid this, the X- and Y-coordinates of all vertices are divided by two. That way, each source tile covers a fourth of the new tile. Additionally, the vertices of the source tiles that are not located in the bottom left of the encompassing tile will gain an offset of 16383 to one of both of the X- and Y-axes. By performing these transformations, the vertices are now in the correct locations. However, as it currently stands, the inner edge vertices of the source tiles create a cross through the center of the new tile where the source tiles would usually line up. To mitigate this, all inner edge vertices are removed. As the vertices were virtually all interpolated from actual points, removing them does not impact the overall shape of the mesh. Removing these points allows us to reduce the file size of the new tile somewhat. Up until this point, the vertices are in their correct locations, and redundant vertices have been removed. Now comes the question of how many of the remaining vertices are to be removed, as keeping all of them would increase the overall resolution of the new mesh significantly. The point of using a TMS-based system is to be able to switch out tiles depending on the level of zoom to best fit the situation. This means that tiles with a lower zoom cover a larger area but contain less detail. However, to make the transitions between zoom levels less jarring, all tiles, no matter the level of zoom, should contain around the same number of points. Thus, by removing 75% of the source vertices, the resolution of the tile remains the same as the source tiles, but the area that it covers is four times as large.

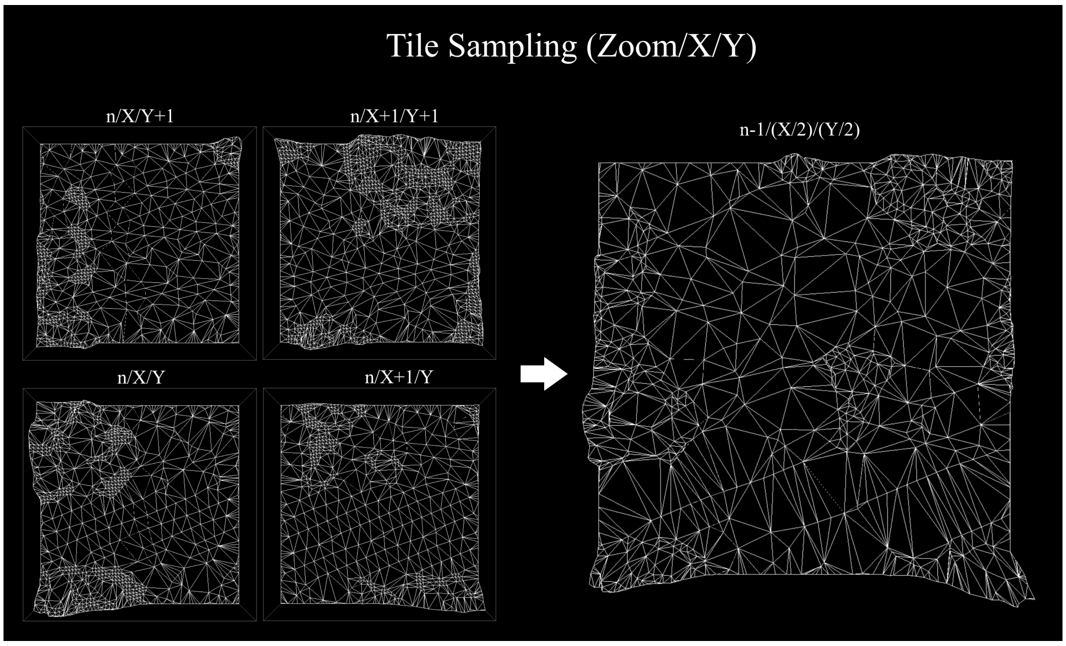


Figure 16. Sampling of higher detail tiles.
Tiles displayed in debug viewer.

5.5 Mesh optimization (TIN)

A *Triangulated Irregular Network*(TIN) is a mesh consisting of irregularly spaced points where vertices that do not provide much additional information to the overall shape are discarded. As for why this is relevant to this particular project, it primarily comes down to reducing the file size. When converting large areas of terrain into layers of QuantizedMesh tiles, up to hundreds of thousands of files are created. Even small reductions in the file size of each tile can have great impacts on the overall size. Reducing the original mesh to a TIN mesh allows us to drastically lower the overall file size while still maintaining the overall shape of the geometry as well as possible (Fig. 17). How much the file size of each tile is reduced depends on the geometry of the mesh. If the mesh contains many small variations in the terrain, then the reduction will not be as significant as when the mesh consists of larger flatter areas. Overall, the average reduction in size that this converter is able to achieve, is around 73% (Table. 7), which is quite significant. An additional benefit of reducing the mesh is that it further speeds up the processes that follow the mesh creation stage. When there are fewer vertices and indices to process, it leads to major speedups in the creation, encoding, and writing of QuantizedMesh tiles. In this case, a speedup of around 32% was achieved.

	Non-optimized mesh	**TIN mesh**	**Reduction (%)**
Time	67836 ms	46067 ms	≈ 32%
size	292 MB	79 MB	≈ 73%

Table 7. Differences in performance and size between a regular mesh and a TIN mesh. The experiment was run on the same 4 adjacent input data tiles producing 4431 unique QuantizedMesh tiles covering six layers (15 to 10). **Time** is the how long the converter took to produce the tiles, while **Size** is the size of the resulting tiles on disk.

The goal is to remove vertices that provide little to no additional information to the shape of the mesh. The first aspect that needs to be addressed, is which properties of the mesh can be utilized to decide the value of any given vertex. When displaying a flat area, there is no need to use a dense mesh, as the shape is simple and there is no obvious visual improvement gained from increasing the number of vertices in the mesh. What makes a shape complex are all the sharp edges and crevices present in the shape. Therefore, by finding the triangles that make up the sharp edges in the mesh, it is possible to flag the points that make up these triangles as being important to the overall shape of the geometry. Points that are not flagged can freely be removed from the mesh, as they are deemed to not contribute much to the overall shape. The method for finding these important triangles is to compare the angle between the surface normals of the triangles and their neighbors. This is called the *dihedral angle*. A surface normal is a vector that is perpendicular to a plane, in this case, the triangular plane. If the angle between the normals is above a certain specified threshold, it means that the two triangles create an edge or crease that is deemed as having great impact on the shape of the mesh. The points of both triangles are then flagged as valuable and should not be removed during future pruning.

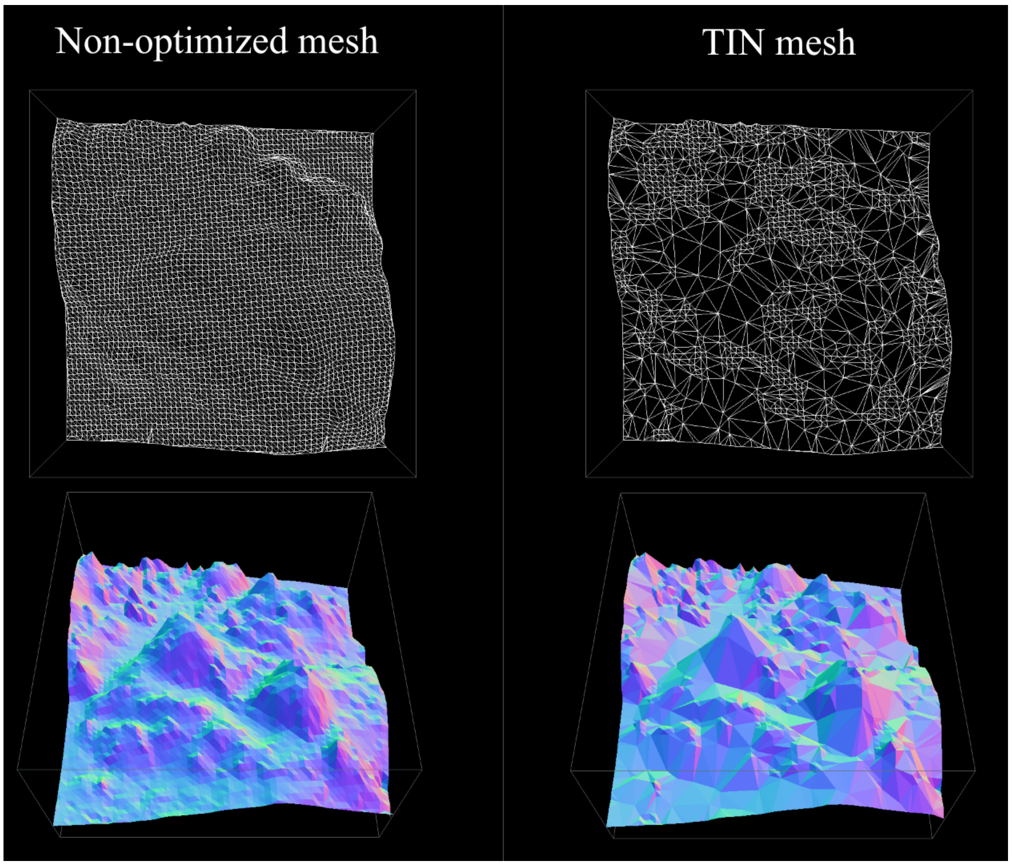


Figure 17. Point density difference.

Visualizing the point density between a non-optimized mesh and a TIN mesh of the same area.

In essence, calculating the surface normal amounts to calculating the cross product between two vectors that make up the plane. The cross product (or vector product) is a vector that is perpendicular to two vectors that exist on the plane. As the rule $\vec{A} * \vec{B} = -\vec{B} * \vec{A}$ states, there is a need to differentiate which vector is used as the surface normal. In this case, the vector extending outward is chosen. The only other requirement applied to these vectors is that they should be *linearly independent*. This means that the vectors cannot have the same nor the opposite direction of one another.

First, both plane vectors need to be defined. One method to ensure that the vectors are linearly independent is to use a common point as their origin and subtract that point from some other point on the plane. Naturally, the points that are used to calculate these starting vectors are the points that define the corners of the plane. Once these vectors have been defined, the cross product is calculated in order to create the surface normal (Alg. 2).

```
function CalculateSurfaceNormal(Triangle)
    $VectorU \leftarrow Triangle.point.B - Triangle.point.A$
    $VectorV \leftarrow Triangle.point.C - Triangle.point.A$
    $Normal.X \leftarrow (VectorU.Y * VectorV.Z) - (VectorU.Z * VectorV.Y)$
    $Normal.Y \leftarrow (VectorU.Z * VectorV.X) - (VectorU.X * VectorV.Z)$
    $Normal.Z \leftarrow (VectorU.X * VectorV.Y) - (VectorU.Y * VectorV.X)$
    return $Normal$
end function
```

Algorithm 2.
Calculating the surface normal of a triangle.

As established earlier in section 5.4.2, for each tile that is processed in the Quantized Mesh creation pipeline, each triangle in that mesh is investigated at some point. However, now in addition to interpolating edge and corner points, the surface normal for each triangle is also calculated and compared with the surface normal of each of the triangle's neighbors to find the dihedral angle. If the angle is sharp, then both triangles whose surface normal was used in the calculation are placed into a protected set of triangles. During a later stage of the pipeline when unwanted triangles are pruned, the triangles in the protected set are not processed. As for the pruning stage, while it would be feasible to remove every triangle that is not in the protected set, it requires considerable adjustment of the dihedral angle threshold to produce good results. There is a tendency to remove either too many points or not enough, as the results can differ greatly depending on the input topography. When too many important points are not correctly flagged and therefore removed, the overall shape of the tile can be significantly altered. When too few points are removed overall, while the shape of the tile is kept intact, there are too many redundant points remaining, leading to unnecessarily large file sizes. To mitigate this issue, there is a need for some leeway. If the dihedral angle threshold is set to effectively remove too many points, instead of immediately removing them, the pruning routine can be configured to remove each point with e.g., a 90% probability. While this will ultimately remove most unprotected points, it results in a somewhat uniform distribution of points throughout areas of the tile that would otherwise be completely empty. This leaves slight variations in flatter areas with points that would have been regarded as insignificant by the angle check, which gives a more natural look to the terrain. However, as this means that the mesh contains a few more points, the file size is also slightly larger. While it is of course up to the user to decide if this tradeoff is worth it, largely, the increase in fidelity is worth it for most, given the small cost.

Chapter 6

Algorithm scalability analysis

6.1 Overview

While performance was an important aspect during development, the main issue to solve was the memory usage scalability of the algorithm. It should be capable of handling vast amounts of elevation data, which, in order to not run out of memory, would require that the algorithm maintains minimal state. To eliminate the need to maintain state, the algorithm should be indifferent to the data that it is processing. It should therefore be possible to process input data in an arbitrary order and still produce the same results. With these rules in place, input data can be loaded, processed, and discarded without regards to other data, eliminating the need to maintain this type of information. With these considerations in mind, the time complexity of the algorithm would scale according to the total number of points processed, given that the terrain complexity remains somewhat similar throughout the data set. Memory usage remains constant during processing.

6.2 Data collection

The algorithm presented in this thesis consists of many parts, but can generally be segmented into three different areas of processing. Namely, segmentation, mesh creation, and tile completion (using the results of the other steps to producing the tile). The mesh creation phase contains many different sections itself, such as triangulation, TIN optimization, and normalization. Collecting the necessary data required for performance analysis entailed measuring the elapsed time for these tasks for each produced QuantizedMesh tile. All measurements are then combined to find the average for each measurement. In this case, finding the average was preferred as the vast majority of tiles contain roughly the same number of points. The tests included producing tile levels 15 to 10 using increasingly larger samples of input data tiles from the same set of tiles. While these tiles were quite similar to each other in terms of size and terrain, there were some small discrepancies which lead to slight variations in the test results. Therefore, the results were scaled to the average number of points per tile across all tests in order to reduce these variations as much as possible.

6.3 Results

Using samples of 6, 12, 25, 50, and 100 input data tiles, the total processing time is shown to grow linearly (Fig. 18). This is to be expected, as the creation of each tile is its own independent action, given that the input data tiles are roughly of similar size. What these results show is that depending on how much data is provided to the algorithm, the execution time grows in proportion.

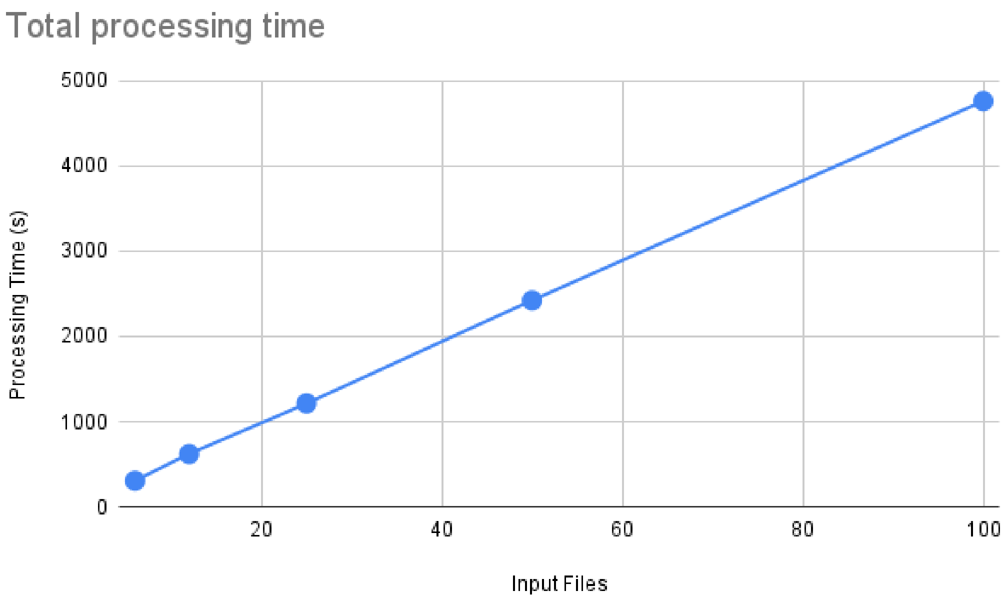


Figure 18. Total processing time of the QuantizedMesh creation algorithm.

The average tile processing times show where the algorithm spends most of its time. The three sections that stand out are triangulation, TIN optimization, and tile completion (Fig. 19). Triangulation and TIN optimization depend on the density of points in the tile. With high point density, triangulating the mesh takes longer. If the mesh is larger and more complex as a result of high density, then TIN optimization takes longer. Tile completion depends on how many points the TIN mesh contains, which varies depending on the terrain of the tile (constant terrain variations lead to fewer points being removed from the mesh during optimization).

As the results show, the average processing time remains roughly the same as the amount of input data increases (Fig. 19). This further proves that processing individual tiles is an independent operation not influenced by any outside factors.

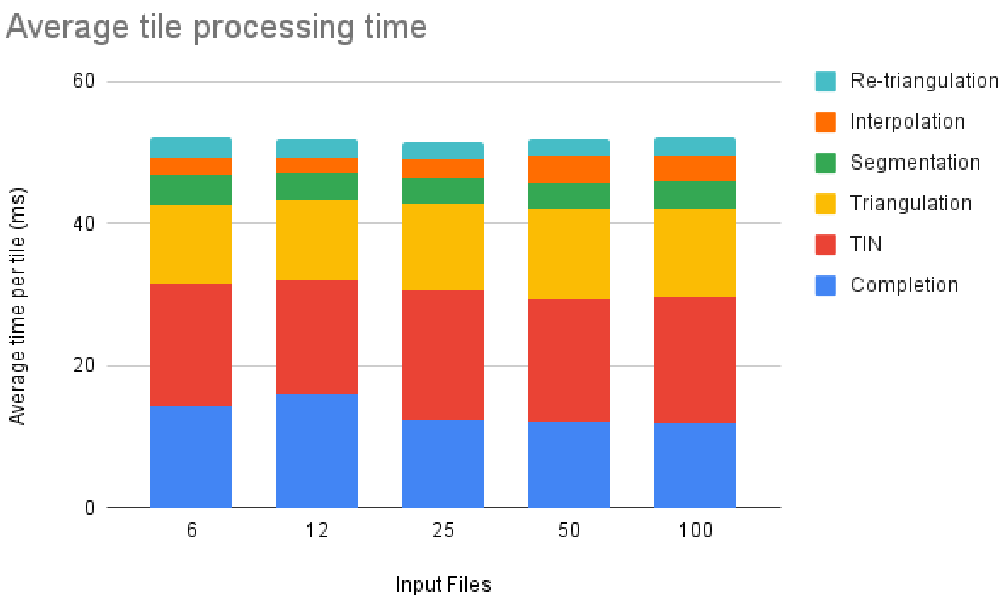


Figure 19. Average processing time of sections in the QuantizedMesh creation algorithm.

Chapter 7

Discussion

This chapter will serve as a discussion about the topics covered throughout this thesis. The main focus will be on the more challenging problems and their solutions that were encountered during the development of the project. To begin the discussion, let us reiterate what the purpose of the project was, its requirements, and if those requirements were fulfilled.

The purpose was to create a software tool that is able to convert large amounts of altitude data into a format more viable for visualization and analysis. As the resulting software tool does indeed produce this data correctly, it can be confidently stated that this requirement has been fulfilled. However, why was this tool needed in the first place? While there are readily available tools for producing a myriad of different geographical data formats, they usually require the user to provide data in specific formats and coordinate systems. Additionally, most of the available tools that are able to produce QuantizedMesh tiles are cloud-based. This is unfortunately not viable, as the converter should be able to work in environments that are restricted in terms of network accessibility. These factors (among others) considered, makes it more beneficial to create an in-house tool for the task. While an in-house tool has more initial costs, and requires further maintenance in the future, it also provides more control. If there is ever a need in the future to support a new file format or to implement a new feature, it can be achieved considerably faster in comparison to requesting said functionality from a third party.

This project proved to be quite challenging, both in terms of design and implementation. The requirement which became the main challenge during this project, due to being quite complex and labor-intensive, was the elimination of gaps between tiles due to the input data not being aligned with the tile grid. The essence of any solution to the problem was to interpolate points using the provided data to keep the edges straight. In this project, the chosen solution was to interpolate the points by finding the intersection between lines crossing the edge of the tile. These lines were defined by linking together points on both the inside and outside the tile. As for the corners, the points were generated by calculating the z-coordinate of a point on a triangular plane at the corners, given the x- and y-coordinates. While this solution managed to produce some good results, a great deal of uncertainty remained. Is this solution fast or slow? Could the same results have been produced using a simpler method? Another solution to the problem, which was theorized by Jan Westerholm, the supervisor of this thesis, was to deal with the issue of misaligned data by interpolating coordinates to fit the tile axis first, and then further process the coordinates into tiles. Using that method, the process of segmenting coordinate data into the correct tiles could have been much faster, as coordinates would not need to be checked one by one to verify which tile they belonged to. Instead, they could be segmented in bulk, leading to a potential performance gain. Additionally, the implementation would be easier to understand for someone not as familiar with the ins and outs of this project, as some steps in the pipeline could be removed. The drawback is that it would use more memory, as it needs to keep multiple input data files loaded to fully cover the area of the tile currently being processed. Additionally, unless tiles were processed in an efficient order with some predictive caching mechanism, thus a minimal need to constantly load new data, an additional overhead caused by file reading operations would be introduced. The takeaway of this would be that while both methods have their fair share of advantages and disadvantages, combining some aspects of each method could produce overall better results, mainly in terms of performance. However, by the time of writing this thesis, there is unfortunately no definitive answer as to which method or combination of methods would produce the best results, as there is no previously established reference implementation.

Chapter 8

Conclusion

Maps are an invaluable tool in modern society and have been for thousands of years. While all maps provide some geographical information, some types of maps are unable to convey the required information accurately enough depending on the use case. In this particular situation, the requirement is primarily to be able to perform calculations using altitude at varying scales. However, the desired format to be used for this purpose is unfortunately not distributed by local land surveying organizations. Additionally, for the data that is distributed, there are no readily available tools for converting said data to QuantizedMesh tiles. Therefore, the solution was to build this tool from the ground up, in order to transform the available altitude data into tiles.

When starting the project, it was vital to gain an adequate understanding of map projections, coordinate systems, and how they relate to one another. This knowledge was very important, as it enabled thinking about the problem at hand much more freely and allowed for designing the application around these requirements and restrictions. It also aided in understanding the thought behind the different parameters of the tile-based QuantizedMesh format. As for the format itself, it proved to be a somewhat involved process to implement the encoder and decoder. This was mainly due to understanding and implementing the different compression algorithms in use in the format and the meticulous verification of the results. However, as this stage of the project was completed, it became clear what kind of data is needed and how it needs to be structured in order to properly produce tiles. These requirements played a significant role in the design of the project.

Designing the pipeline for how raw input data is processed and transformed into QuantizedMesh tiles of different levels was quite complicated, but proved to be a good exercise in problem-solving. It mainly required identifying the core challenges of the problem and dividing them into smaller tasks that could be isolated and implemented separately. Initially, the main issue proved to be the processing of massive amounts of data too large to fit into working memory all at once. Data from one file could be spread over multiple output tiles which themselves may contain data found in other input data files. As the files cannot all fit in working memory at once, it was not possible to create tiles from one unified source data set. Therefore, the implemented solution was to handle input files one by one, but merging data of those output tiles whose content was not mutually exclusive. Finally, these generated output tiles would then be used as a source to sample from when creating the less detailed tile layers.

However, this was not the most challenging problem of the project. That award goes to eliminating gaps between tiles, which turned out to be substantially more complicated than anticipated. In this project, the method to eliminate gaps contains a several steps. First, we gather points that exist inside and right outside the borders of the tile. Secondly, we use those points to generate a mesh that extends out past the bounds of the tile. Third, we find all points of intersection between the borders of the tile and the mesh, discarding all other points outside the bounds of the tile. Finally, we use that collection of points to create the final version of the tile, having eliminated any occurrences where the edge of the tile mesh did not line up with the border of the tile bounds.

The different aspects of these problems could all be further divided into seemingly smaller and smaller tasks. This ended up providing a greater understanding of the problem at hand, as well as a clearer path through the implementation phase of the project. However, the solutions to these problems were not clear from the start, and even when the actual ideas became clearer, they still required a fair number of iterations to properly apply. Overall, the largest amount of time was spent iterating upon and tweaking the gap elimination and mesh optimization passes of the pipeline. While the current result is quite satisfactory, there could still be some improvements in quality and efficiency, e.g., further reducing redundant points in large flat areas such as bodies of water.

Although the design and implementation of the project required considerable thought and many iterations, the result was satisfactory. The project was successful in performing the task as per its requirements. While challenging, the design and implementation phase of the project provided valuable hands-on experience in building a complicated piece of software from the ground up.

Chapter 9

Generering av kvantiserade nät - Svensk sammanfattning

Kartor är ett ovärderligt verktyg i dagens samhälle och har varit det i tusentals år. Även om alla kartor förmedlar någon slags geografisk information, klarar vissa typer av kartor inte av att förmedla den efterfrågade informationen tillräckligt noggrant beroende på användningsfallet. I det här specifika fallet är kravet främst att kunna utföra beräkningar med hjälp av höjddata på olika skalor. Tyvärr är det önskade formatet för detta syfte inte tillgängligt på lokala lantmätningsorganisationer. Dessutom finns det inga tillgängliga verktyg för att konvertera dessa data till QuantizedMesh-rutor. Lösningen var därför att bygga detta verktyg från grunden för att producera rutorna med hjälp av tillgängliga höjddata och möjligtvis övriga format såvida behovet uppstår.

När projektet att designa och utveckla detta verktyg inleddes var det viktigt att få en tillräckligt bra uppfattning om kartor, kartprojektioner, koordinatsystem och hur de förhåller sig till varandra. Denna bakgrundskunskap var mycket viktig för att verkligen förstå problemet och designa applikationen med hänsyn till dess krav och begränsningar. Bakgrundskunskapen var även nödvändig för att förstå tanken bakom de olika parametrarna i QuantizedMesh-formatet. Formatet i sig själv är väl upplagt och det enda som till en början var något utmanande var implementationen av dess kodare och avkodare, eftersom många sektioner i formatet använder sig av diverse metoder för datakompression. Efter att detta skede av projektet var färdigt blev det klart vilka data som egentligen behövs för att producera dessa filer, vilket hade stor inverkan på projektets design.

Att komma fram till en pipeline hur källdata bearbetas och omvandlas till QuantizedMesh-filer med varierande detaljnivåer var ganska komplicerat. Det krävde att identifiera problemets grundläggande utmaningar och dela upp dem i mindre uppgifter som kunde isoleras och behandlas separat. Huvudproblemet visade sig vara de enorma datamängderna som måste behandlas. Mängden data som krävs för att representera stora områden, t.ex. länder, är för stor för att samtidigt rymmas i arbetsminnet på en modern arbetsstation. Koordinater från en fil källdata kunde vara utspridda över flera outputfiler som i sig själva kunde innehålla koordinater från andra källdatafiler. Eftersom alla filer inte ryms i arbetsminnet samtidigt var det inte möjligt att producera output filer från en enhetlig datauppsättning. Den slutliga lösningen var att hantera källfilerna en efter en, och att sammanfoga data från de resulterande outputfilerna vars innehåll inte var ömsesidigt uteslutande. Slutligen används dessa output filer som ny källdata när de mindre detaljerade detaljnivåerna skapas.

Detta var dock inte det mest utmanande problemet i projektet. Det visade sig att eliminera glappet mellan QuantizedMesh-rutorna var betydligt mer komplicerat än alla andra problem som påträffades. Metoden som användes för att eliminera glappet innehåller flera olika steg. Först samlar man in punkter som befinner sig precis utanför rutans gränser. Sedan användes dessa punkter i kombination med tidigare insamlade punkter för att generera en mesh som sträcker sig utanför rutans gränser. Därefter ska man hitta alla skärningspunkter mellan meshens och rutans gränser samt kasta bort alla övriga punkter utanför gränserna. Slutligen användes denna samling punkter för att skapa den slutliga versionen av meshen för denna specifika ruta, efter att ha eliminerat alla förekomster där meshens gränser inte ligger i linje med rutans gränser.

De olika delmomenten i dessa problem kunde alla delas upp i allt mindre uppgifter. Detta ledde till att man fick en bättre uppfattning av problemet och en tydligare plan genom implementationsfasen av projektet. Lösningarna till dessa problem var dock inte klara från början, och även efter att idéerna blev tydligare krävdes det fortfarande en hel del iterationer för att kunna tillämpa dem på rätt sätt. Totalt gick största delen av tiden åt till att iterera och justera algoritmerna för eliminering av glapp och optimering av meshen. Även om de nuvarande resultaten är ganska tillfredsställande, är det fortfarande möjligt att förbättra både kvaliteten och effektiviteten av algoritmerna. Speciellt på stora flata områden som vattenmassor är det möjligt att vidare reducera antalet redundanta punkter.

Även om projektets design och implementation krävde mycket genomtanke och många iterationer, var resultatet tillfredsställande. Projektet var framgångsrikt i sin uppgift att utföra den specificerade uppgiften enligt kraven till en bra nivå. Processen visade sig även vara mycket givande i och med att den möjliggjorde mycket värdefull praktisk erfarenhet att designa och implementera en komplicerad applikation från grunden.

References

[1] B. Delaunay. “Sur la sphère vide. A la mémoire de Georges Vorono¨■. In: *Bulletin de l’Académie des Sciences de l’URSS. Classe des sciences mathématiques et na* 6 (1934), pp. 793–800. URL: http://mi.mathnet.ru/im4937.

[2] *The Universal Transverse Mercator (UTM) Grid.* 2001. URL: https://pubs.usgs.gov/fs/2001/0077/report.pdf.

[3] V. Domiter and B. Žalik. “Sweep-line algorithm for constrained Delaunay triangulation”. In: *International Journal of Geographical Information Science* 22.4 (2008), pp. 449–462. URL: https://doi.org/10.1080/13658810701492241.

[4] *Implementation Practice Web Mercator Map Projection.* National Geospatial-Intelligence Agency (NGA), 2014. URL: https://web.archive.org/web/20141009142830/http://earth-info.nga.mil/GandG/wgs84/web_mercator/(U)%20NGA_SIG_0011_1.0.0_WEBMERC.pdf.

[5] Heather Smith. *Geographic vs Projected Coordinate Systems.* 2020. URL: https://www.esri.com/arcgis-blog/products/arcgis-pro/mapping/gcs_vs_pcs/.

[6] URL: http://dirsig.cis.rit.edu/docs/new/coordinates.html.

[7] URL: https://cgrsc.ca/resources/geodetic-reference-systems/ellipsoidal-coordinate-system/.

[8] *About Cesium.* URL: https://cesium.com/about/.

[9] Paul Bolstad. *GIS Fundamentals, 6th Edition.* Chap. 3. Geodesy, Datums, Map Projections, and Coordinate Systems. URL: https://www.dropbox.com/s/t00mopkay3jdsrh/Chapter3_6th_small.pdf?dl=0.

[10] *Earth Gravitational Model 1996.* URL: https://cddis.nasa.gov/926/egm96/egm96.html.

[11] The Editors of Encyclopaedia Britannica. *Projection - cartography.* URL: https://www.britannica.com/science/projection-cartography.

[12] *Mercator projection.* URL: https://www.britannica.com/science/Mercator-projection#/media/1/375638/231099.

[13] *OpenGL Vertex Specification*. URL: https://www.khronos.org/opengl/wiki/Vertex_Specification.

[14] *Quantize Definition*. URL: https://www.merriam-webster.com/dictionary/quantize.

[15] *quantized-mesh-1.0 terrain format*. URL: https://github.com/CesiumGS/quantized-mesh.

[16] *Theoretical mean water and geodetical height systems in Finland*. URL: https://en.ilmatieteenlaitos.fi/theoretical-mean-sea-level.

[17] *Wavefront .obj description*. URL: https://www.loc.gov/preservation/digital/formats/fdd/fdd000508.shtml.

[18] Eric Weisstein. *Vertex*. URL: https://mathworld.wolfram.com/Vertex.html.

[19] *What Is a View Frustum*. URL: https://learn.microsoft.com/en-us/previous-versions/windows/xna/ff634570(v=xnagamestudio.42).