Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

3D Terrain with Level of Detail

Written report for the module
BTI3041 – Project 2
by

Amar Tabakovic

Bern University of Applied Sciences
Engineering and Information Technology
Computer Perception & Virtual Reality Lab

Supervisor
Prof. Marcus Hudritsch

January 19, 2024

Abstract

Rendering terrains is a central task for video games, geographic information systems and simulation software, but also computationally expensive. Optimizations, one of which is the level of detail (LOD), are necessary in order to ensure adequate performances. Numerous algorithms were developed in the last 30 years which tackle the problem of efficient terrain rendering. A demo terrain renderer is developed with the goal to demonstrate and compare some of the ideas of these algorithms. The main implemented algorithm is mostly based on GeoMipMapping [dB00], but also draws inspiration from GPU-based Geometry Clipmaps [AH05] and other approaches. The implementation was tested on a 14000×14000 heightmap of Switzerland and bordering regions, yielding around 60 FPS on a 2020 MacBook Air while delivering decent visualizations.

Contents

1 Introduction **8**
- 1.1 Goals of this Project . . . 8
- 1.2 Intended Readership . . . 8
- 1.3 Notation and Terminology . . . 9
  - 1.3.1 Mathematical Notation . . . 9
  - 1.3.2 The Term "LOD Level" . . . 9
- 1.4 Outline of the Report . . . 9

2 Basics of Terrain Rendering **10**
- 2.1 Terrain Data Representation . . . 10
  - 2.1.1 Heightmaps . . . 10
  - 2.1.2 Triangulated Irregular Networks . . . 11
- 2.2 Bintrees and Quadtrees . . . 12
- 2.3 View-frustum Culling . . . 12
- 2.4 Potential Problems During Terrain Rendering . . . 14
  - 2.4.1 Cracks . . . 14
  - 2.4.2 Popping . . . 15

3 Existing Work and Literature **16**
- 3.1 Algorithms and Approaches for Terrain LOD . . . 16
  - 3.1.1 ROAM . . . 17
  - 3.1.2 GeoMipMapping . . . 18
  - 3.1.3 (GPU-based) Geometry Clipmaps . . . 19
  - 3.1.4 Concurrent Binary Trees . . . 20
  - 3.1.5 Conclusion . . . 21
- 3.2 Terrain LOD in Real-world Systems . . . 22
  - 3.2.1 Game Engines . . . 22

4 ATLOD: A Terrain Level of Detail (Renderer) **25**
- 4.1 Used Technologies . . . 25
- 4.2 Basic Setup and Architecture . . . 26
  - 4.2.1 Overview . . . 26
  - 4.2.2 Command Line Arguments . . . 26
  - 4.2.3 Shaders . . . 27
  - 4.2.4 Camera . . . 27
  - 4.2.5 Skybox . . . 29
  - 4.2.6 Heightmaps . . . 29

4.2.7 Base Terrain 30
4.3 Naive Brute-force Algorithm 31
4.3.1 Vertex and Index Organisation 31
4.3.2 Rendering 32
4.4 GeoMipMapping 33
4.4.1 Class Structure 34
4.4.2 Blocks 34
4.4.3 Vertex and Index Organisation 35
4.4.4 Rendering 42

5 Results **48**
5.1 Experimental Setup 48
5.1.1 Hardware 48
5.1.2 Height Data and GeoMipMapping Configuration 48
5.1.3 Benchmarks 49
5.2 Performance Benchmarks 50
5.2.1 Flyover from Corner to Corner 50
5.2.2 360° Rotation 50
5.3 Visual Accuracy Benchmarks 50
5.3.1 Large Terrain Screenshots 51
5.3.2 Low FOV Screenshots 51
5.4 Memory Consumption 51
5.4.1 RAM 51
5.4.2 GPU Memory 51
5.4.3 Examples 52

6 Discussion **53**

7 Conclusion **54**
7.1 Potential Improvements 54
7.2 Outlook for the Bachelor Thesis 54

Bibliography **58**

A DEM Preprocessing **59**

B Visual Accuracy Benchmarking Images **60**
B.0.1 Large Terrain Screenshots 60
B.0.2 Low FOV Screenshots 65

List of Tables

5.1 The specifications of the used MacBook Air 2020. 48
5.2 Rendering settings for the benchmarks. 49
5.3 RMSE of the large terrain screenshots 1 to 5. 50
5.4 RMSE of the large terrain screenshots 1 to 5. 50
5.5 RMSE of the large terrain screenshots 1 to 5. 51
5.6 RMSE of the low FOV screenshots 1 to 5. 51
5.7 Memory consumption by the vertex and index buffers for different block sizes. 52
5.8 Memory consumption by the heightmap texture on the GPU for various heightmap sizes. 52
5.9 Memory consumption by the block list at different block sizes and heightmap sizes. 52

List of Figures

2.1 2000 × 2000 heightmap of the mountain Dom in Valais, Switzerland retrieved from SwissTopo [Fed]. 11
2.2 Example of a TIN. Note that the left area represents a terrain area with many changes (e.g. mountains, hills, etc.), and the right area represents an area with few changes (e.g. flat areas). . 11
2.3 Example of a bintree (a) and a quadtree (b). 12
2.4 Example of a view-frustum 13
2.5 Example of a terrain block with its AABB defined by $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$, marked in red. 13
2.6 Example of view-frustum culling with a quadtree viewed from the top. The view-frustum is marked in yellow and blocks that intersect the view-frustum are marked in green. 14
2.7 Illustration of a crack (a) and some examples of cracks in a real rendered terrain (b). 15

3.1 Top-down view of an example triangulation generated by ROAM (taken from [DWS+97]). 17
3.2 Example of each GeoMipMap of a 5 × 5 block. The omitted vertices of lower LOD GeoMipMaps are marked as dotted circles (based on [dB00]). 18
3.3 Example of GeoMipMapping's crack avoidance between a LOD 2 and a LOD 1 GeoMipMap of two 5 × 5 blocks (based on [dB00]). 19
3.4 Example of the flat mesh in Geometry Clipmaps with $n = 15$, $m = 4$ and $l = 3$ (based on [AH05]). 20
3.5 Unreal Engine's landscape system LOD morphing (taken from [Gam]). 23

4.1 The default sunset gradient skybox. 29
4.2 Example of a terrain layout for triangle strips. The looping index `i` goes from 0 to the terrain height and `j` from 0 to the terrain width. The final indices to be rendered are 0, 3, 1, 4, 2, 5, `RESTART`, 3, 6, 4, 7, 5, 8, `RESTART`. 31
4.3 Example of a normal vector calculation of the bottom right face. The same process gets repeated for the other three adjacent faces. 32
4.4 The index buffer organisation of the single flat block. The variable n corresponds to the maximum LOD level. 35

4.5 Every possible border permutation for a LOD 2 GeoMipMap of a 5×5 block. The center subblocks have been omitted from the illustration. 36
4.6 Illustration of accessing the start index and size of the subsets of the index buffer for LOD 1 and border permutation $(0, 0, 0, 0)$ . . 37
4.7 Illustration of a flat terrain showcasing the linearly growing distance mode (a) and exponentially growing distance mode (b). The red, green and blue colors indicate successively lower LOD levels, starting from the maximum level in the center. 43

5.1 The 13922×14140 16-bit greyscale heightmap used for benchmarking (retrieved from OpenTopography [NAS13]). In this figure, the gray values were converted from 0,...,65535 to 0,...,255 in order to make the heights more visible. 49

B.1 Screenshot showcasing the screenshot of a large section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b), and the binarised absolute difference (d) of (c). The computed RMSE is 3.94. 60
B.2 Screenshot showcasing the screenshot of a large section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b), and the binarised absolute difference (d) of (c). The computed RMSE is 3.1. 61
B.3 Screenshot showcasing the screenshot of a large section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b), and the binarised absolute difference (d) of (c). The computed RMSE is 2.59. 62
B.4 Screenshot showcasing the screenshot of a large section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b), and the binarised absolute difference (d) of (c). The computed RMSE is 1.96. 63
B.5 Screenshot showcasing the screenshot of a large section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b), and the binarised absolute difference (d) of (c). The computed RMSE is 2.32. 64
B.6 Screenshot showcasing the screenshot of a small section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b) and the binarised absolute difference (d) of (c). The FOV is set to 6° and the computed RSME is 4.82. . . 65
B.7 Screenshot showcasing the screenshot of a small section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b) and the binarised absolute difference (d) of (c). The FOV is set to 3° and the computed RSME is 5.71. . . 66
B.8 Screenshot showcasing the screenshot of a small section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b) and the binarised absolute difference (d) of (c). The FOV is set to 2° and the computed RSME is 4.78. . . 67

B.9 Screenshot showcasing the screenshot of a small section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b) and the binarised absolute difference (d) of (c). The FOV is set to 1° and the computed RSME is 5.3. . . 68
B.10 Screenshot showcasing the screenshot of a small section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b) and the binarised absolute difference (d) of (c). The FOV is set to 1° and the computed RSME is 4.49. . . 69

Listings

4.1 The fog calculation in the fragment shader. 33
4.2 Method `GeoMipMapping::loadVertices()` that generates the vertex array object and loads the vertex buffer with the flat mesh of size $blockSize \times blockSize$ centered around $(0, 0, 0)$. 37
4.3 Method `GeoMipMapping::loadTopLeftCorner()` which loads in the indices of the top left corner of the flat mesh for a given LOD level and border permutation 39
4.4 Method `GeoMipMapping::determineLodDistance()` that determines the LOD level of a block based on its distance to the camera. 43
4.5 The method `GeoMipMapping::calculateBorderBitmap()` which computes the border permutation bitmap for a given block. . . . 44
4.6 The two draw calls occuring inside the second loop over all blocks inside the method `GeoMipMapping::render()`. 44
4.7 The vertex shader of the GeoMipMapping implementation. . . . 45
4.8 Calculating the normal vector in the fragment shader using the heightmap texture. 46

Chapter 1

Introduction

In 3D computer graphics, rendering is the central task. Many practical applications of 3D computer graphics make use of terrains, such as flight simulators, open-world video games, and Geographic Information Systems (GIS) [LWC+02, p. 185]. At the same time, rendering terrains, which are large and constantly visible, is computationally expensive and optimizations are necessary in order to ensure adequate performance.

One area which offers potential for optimizations is the *level of detail (LOD)*. The concept of LOD is based on the intuitive idea that the farther away an object is, the fewer details are going to be visible to the human eye. Over the last three decades, numerous algorithms and approaches have been published for the problem of efficient terrain rendering.

1.1 Goals of this Project

The primary goal of this project is the study and exploration of terrain rendering algorithms. First, the basics of terrain rendering are studied and an overview of the state-of-the-art of terrain rendering is constructed. Afterwards, a demo terrain renderer is developed, using the ideas from one or more of the evaluated algorithms.

This project is restricted to simple heightmap rendering, without any real-time streaming/paging of terrain data.

1.2 Intended Readership

The reader is assumed to be familiar with the basics of computer graphics, C++ and OpenGL.

1.3 Notation and Terminology

1.3.1 Mathematical Notation

This report uses the following mathematical notation:

- The coordinate system is a right-handed coordinate system with y as the up-direction, unless explicitly stated otherwise.
- $\mathbb{N}$ denotes the set of natural numbers, $\mathbb{R}$ is the set of real numbers, $\mathbb{R}^n$ is the set of real numbers in n dimensions.
- $\mathbf{p} = (p_x, p_y, p_z)$ denotes a point in $\mathbb{R}^3$.
- $\mathbf{v} = (v_x, v_y, v_z)$ denotes a vector in $\mathbb{R}^3$.
- $\mathbf{M}$ denotes a matrix in $\mathbb{R}^{n \times n}$.

1.3.2 The Term "LOD Level"

The definition of what "LOD level 0" and what "LOD level l" (l = maximum LOD) mean is different from paper to paper. Normally, LOD systems use 0 for the highest resolution and l for the lowest resolution. In this report, the opposite (and slightly more intuitive) approach is followed: l denotes the highest resolution, and 0 denotes the lowest resolution.

1.4 Outline of the Report

This report is structured as follows:

- Chapter 2 introduces the reader to the basics of terrain rendering. The topics covered include terrain data representation, common optimizations and potential problems during rendering.
- Chapter 3 gives an overview of the state of the art of terrain rendering. Various algorithms and their central ideas are presented in a high-level manner. Afterwards, some examples of real-world systems using terrain LOD algorithms are listed.
- Chapter 4 describes the demo terrain renderer (named ATLOD) which was developed. The basic features and the details of the implemented algorithm are presented.
- In chapter 5, the performance and visual accuracy of ATLOD is measured and the results documented.
- Chapter 6 gives a short discussion of the results from chapter 5.
- Chapter 7 concludes this report by mentioning some potential improvements and the outlook for the bachelor thesis.

Chapter 2

Basics of Terrain Rendering

2.1 Terrain Data Representation

2.1.1 Heightmaps

One way of representing terrains is using *heightmaps*. A heightmap is a $n \times n$-grid that contains the height value y for each (x, z)-position. Positions are always spaced evenly in a grid-like manner, but the distance between any two neighboring positions (in other words the (x, z)-scale) is variable.

The main advantage of heightmaps is that they allow for very simple storage and manipulation of height data, e.g. in form of images, where low color values represent low areas of terrain and vice versa for high color values. The color representation of an image influences the number of possible height values:

- For an 8-bit grayscale image, 256 height values are supported.
- For a 16-bit grayscale image, 65536 height values are supported.
- For an 8-bit RGB image, more than 16 million height values are supported.

Retrieving the height value for a given (x, z)-position is easy, which consists of a simple lookup at the given position in the image. Figure 2.1 shows a 2000×2000 heightmap of the mountain Dom in Valais, Switzerland.

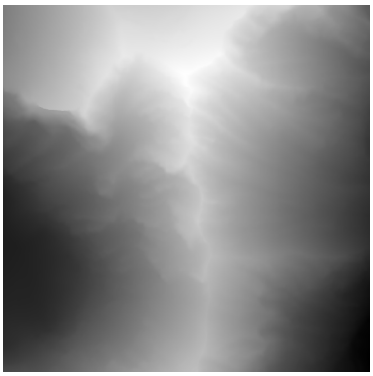


Figure 2.1: 2000×2000 heightmap of the mountain Dom in Valais, Switzerland retrieved from SwissTopo [Fed].

2.1.2 Triangulated Irregular Networks

A less commonly used alternative to the heightmap is the *triangulated irregular network (TIN)* data structure. A TIN consists of a collection of 3-dimensional vertices, where the arrangement of vertices can be irregular. Figure 2.2 shows an example of a TIN.

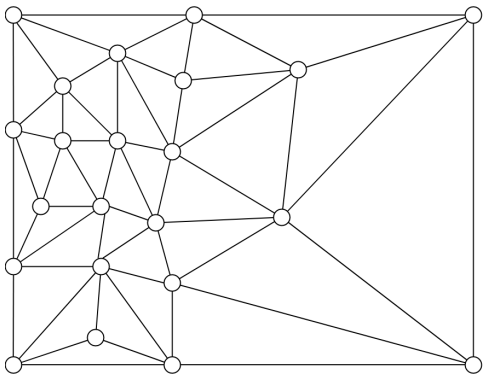


Figure 2.2: Example of a TIN. Note that the left area represents a terrain area with many changes (e.g. mountains, hills, etc.), and the right area represents an area with few changes (e.g. flat areas).

The main advantage of TINs is that fewer polygons need to be used for e.g. smooth terrain areas. Another advantage is that special terrain features can be modelled which are usually difficult to model with heightmaps, such as overhangs, cliffs and caves [LWC+02]. The disadvantage of TINs, however, is that the full (x, y, z)-coordinates need to be stored, whereas with heightmaps, only the height value y needs to be stored. Another disadvantage of TINs is that many terrain LOD algorithms work mainly with heightmaps, such as [DWS+97, dB00, RHSS98, LH04, AH05, Ulr02, Str09, Dup20], and not with TINs.

2.2 Bintrees and Quadtrees

Binary triangle trees (bintrees) and *quadtrees* are recursive data structures based on triangles and quads respectively, and are used to represent the terrain's mesh. Bintrees and quadtrees are mostly found in historical algorithms, such as [LKR+96] and [DWS+97], but have recently been revitalized in [Dup20].

A bintree consists of up to two child triangles, both of which also consist of up to two child triangles each, and so forth. Quadtrees are structured similarly, with a quad consisting of up to four child quads, and each child quad consisting of up to four child quads, and so forth. Figure 2.3 shows an example of a bintree and a quadtree.

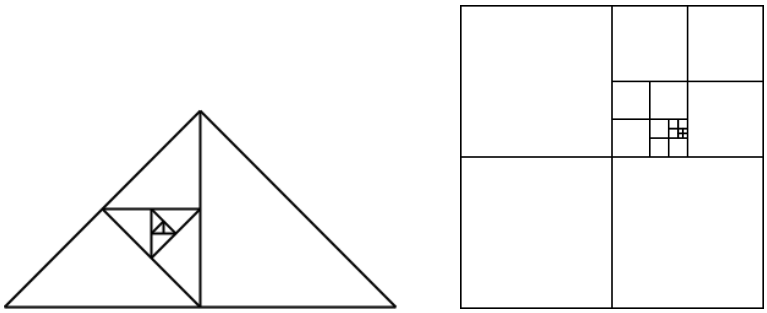


Figure 2.3: Example of a bintree (a) and a quadtree (b).

The main advantage of bintrees and quadtrees is that LOD can be modelled very naturally with them. Bintree/quadtree sections with few children correspond to a low LOD and vice versa for bintree/quadtree sections with many children. Their disadvantage, however, is that they require frequent modification, which is costly.

2.3 View-frustum Culling

View-frustum culling is an optimization technique commonly used in computer graphics. View-frustum culling is used in numerous terrain LOD approaches, such as [LKR+96, DWS+97, dB00, LH04, Str09]. The *view-frustum* is the 3-dimensional pyramid that represents the space that is visible to the camera. It is defined by six planes: the near, far, left, right, top and bottom face. Each face has a normal vector and a distance from the origin. Figure 2.4 shows an example of a view frustum.

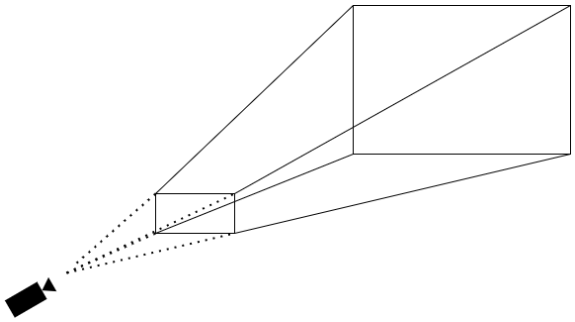


Figure 2.4: Example of a view-frustum

The main idea of view-frustum culling is to check whether the *bounding volume* of an object is contained (at least partially) inside the view-frustum, and if not, to simply not render the object. This dramatically reduces the number of draw calls and the number of vertices that get rendered. The bounding volume of an object is the 3-dimensional volume such that it contains the entire object inside of it. There are different types of bounding volumes, such as *axis-aligned bounding boxes (AABB)*, *oriented bounding boxes (OBB)*, *bounding spheres*, and more. AABB's are commonly used in terrain LOD algorithms [dB00, LH04, Str09] and are defined with two points $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$, which indicate both endpoints of the AABB. Figure 2.5 shows a terrain block with its AABB in red.

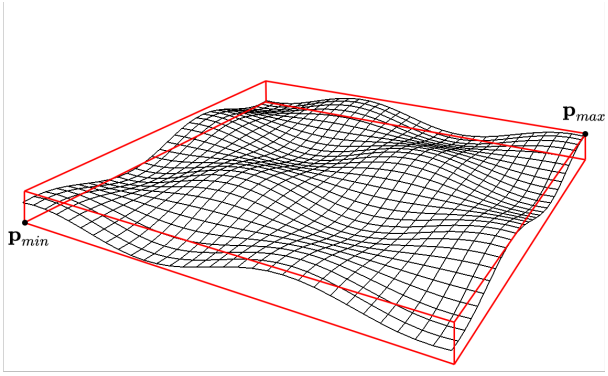


Figure 2.5: Example of a terrain block with its AABB defined by $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$, marked in red.

View-frustum culling can be further optimized by arranging the scene hierarchy (i.e. the terrain hierarchy) into a *space-parititioning data structure*. A widely-used structure is the already previously mentioned quadtree, where leaf nodes contain the renderable terrain sections and where each node has an AABB such that it contains all AABBs of its child nodes. Note that the quadtree in this case is not the same kind of quadtree from the previous section. At render time, the quadtree gets traversed starting from the root node. The intersection of the view-frustum with the AABB of each of the four child nodes gets calculated and if the AABB of a child node intersects with the view-frustum, the child

node gets recursively traversed and the same steps are performed until reaching a leaf node, at which point the terrain section gets rendered. The number of AABB-view-frustum-intersection calculations gets reduced, however at the cost of slightly higher memory consumption. Figure 2.6 shows an example of quadtree-based view-frustum culling.

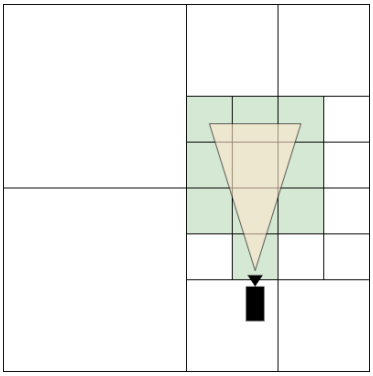


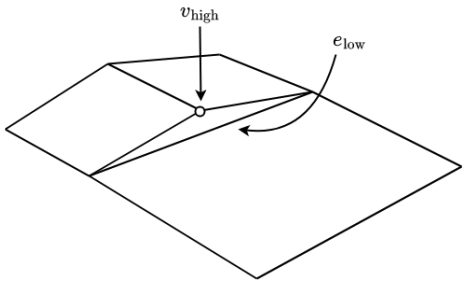
Figure 2.6: Example of view-frustum culling with a quadtree viewed from the top. The view-frustum is marked in yellow and blocks that intersect the view-frustum are marked in green.

2.4 Potential Problems During Terrain Rendering

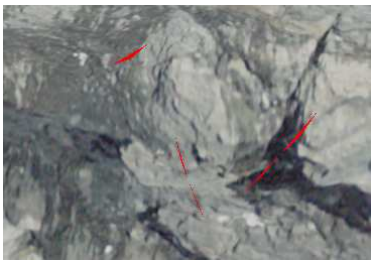
While terrain LOD algorithms dramatically improve the performance of terrain rendering, there are certain issues that can occur.

2.4.1 Cracks

Cracks and holes in terrains can appear when a higher LOD terrain section is bordered by a lower LOD terrain section. The main problem is that when a vertex v_{high} of a higher LOD terrain section lies on the edge e_{low} of a lower LOD terrain section and the y coordinate of v_{high} is greater or less than the height of e_{low} at that point, the difference in height causes the crack to appear, as shown in figure 2.7.



(a) The crack is caused by the height difference of v_{high} and e_{low}.



(b) The background color is set to red to highlight the cracks.

Figure 2.7: Illustration of a crack (a) and some examples of cracks in a real rendered terrain (b).

Cracks can be solved by either of the following, depending on the capabilities of the LOD approach:

- Removing the vertex in question, causing the higher and lower LOD meshes to be connected seamlessly (in figure 2.7 vertex v_{high}).
- Inserting an extra vertex at the border edge of the lower LOD mesh [LWC+02, p. 194] (in figure 2.7 on top of vertex v_{high}). The disadvantage of this is that an extra vertex needs to get created.
- Covering the cracks by rendering a strip at the border of the two meshes [AH05].

2.4.2 Popping

The phenomenon of *popping* occurs when the camera is moving and the change in LOD level causes visual pops to appear. Popping decreases the realism of the terrain and should be as minimal as possible. Popping can be reduced with *vertex morphing* [dB00, LH04, Str09], i.e. by animating the transition of one LOD level to the next seamlessly through interpolation.

Chapter 3

Existing Work and Literature

This chapter starts off by presenting some of the existing algorithms and approaches to terrain rendering. Afterwards, a selection of real-world systems are given, in which terrain LOD algorithms are used.

3.1 Algorithms and Approaches for Terrain LOD

Terrain LOD is a well-researched topic and over the last three decades, numerous approaches have been published. In the following, some of the most important publications are listed in chronological order. The approaches that are described in greater detail in the upcoming subsections are highlighted in **bold**:

- "Real-Time, Continuous Level of Detail Rendering of Height Fields" [LKR+96] by Lindstrom *et al.* in 1996.
- **"ROAMing Terrain: Real-time Optimally Adapting Meshes"** [DWS+97] by Duchaineau *et al.* in 1997.
- "Real-Time Generation of Continuous Levels of Detail for Height Fields" [RHSS98] by Röttger *et al.* in 1998.
- **"Fast Terrain Rendering Using Geometrical MipMapping"** [dB00] by de Boer in 2000.
- "Rendering Massive Terrains using Chunked Level of Detail Control" [Ulr02] by Ulrich in 2002.
- "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids" [LH04] by Hoppe and Losasso in 2004 and the follow-up **"Terrain Rendering Using GPU-Based Geometry Clipmaps"** by Asirvatham and Hoppe [AH05] in 2005.
- "Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD)" [Str09] by Strugar in 2009.

- **"Concurrent Binary Trees (with application to longest edge bisection)"** [Dup20] by Dupuy in 2020.

In the following subsections on the algorithms, all presented ideas are taken from their respective original publications, unless noted otherwise.

3.1.1 ROAM

ROAM (short for **R**eal-time **O**ptimally **A**dapting **M**eshes) is a terrain LOD algorithm developed by Duchaineau *et al.* [DWS+97] published in 1997. ROAM represents the terrain mesh using bintrees and performs triangle splits and merges for generating and removing detail. Figure 3.1 shows an example of a triangulation generated by ROAM.

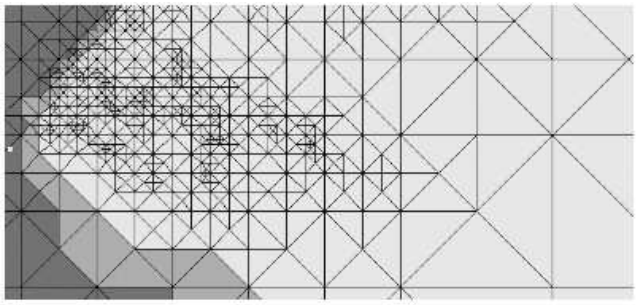


Figure 3.1: Top-down view of an example triangulation generated by ROAM (taken from [DWS+97]).

The central idea of the algorithm is temporal coherence: between two frames, the meshes are often very similar. This means that the mesh from a previous frame can be used to compute the mesh of the current frame, rather than building up the mesh from ground up. This is done using two priority queues: a split queue $\mathcal{Q}_s$ and a merge queue $\mathcal{Q}_m$. The split queue contains splittable triangles T and the merge queue contains mergable triangle pairs (T, T_B). At each frame, the terrain mesh gets split and merged using $\mathcal{Q}_s$ and $\mathcal{Q}_m$. until either the required size/accuracy is reached or the time runs out. The splits and merges always result in a continuous mesh, i.e. the mesh cannot contain any T-junctions. The elements of $\mathcal{Q}_s$ and $\mathcal{Q}_m$ are ordered by various geometric error metrics, some of which are the following:

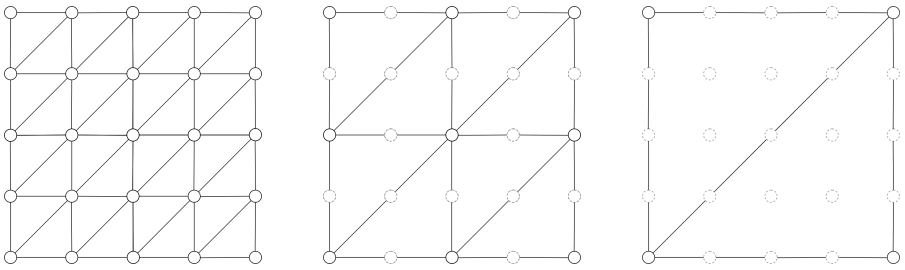
- Nested bounding volumes named *wedgies*, which are defined to include the entire x and z extent of a triangle and its subtriangles plus some padding space above and below the highest and lowest points respectively. Wedgies are computed while building the initial mesh at the beginning of the algorithm.
- Another metric is the geometric screen distortion, i.e. the distance between where a node is supposed to be on the screen and where the algorithm actually places the node. The maximum of all distances is calculated and used as the base priority metric of the algorithm.

While the bintree trees gets traversed, various flags are updated which indicate whether a wedgie is inside the view-frustum completely, partially or not at all and based on these flags, bintree children outside of the view-frustum do not get recursively descended.

3.1.2 GeoMipMapping

Geometrical Mipmapping (GeoMipMapping) is a terrain LOD approach published by de Boer [dB00] in the year 2000. The central idea of GeoMipMapping is its analogy to texture mipmapping: just like how textures of far away objects are rendered using lower resolution texture mipmaps, terrain areas that are far away from the camera should also be rendered with a lower resolution mesh.

This is achieved by splitting up the terrain into so-called *blocks* (also called *patches*) of a fixed side length $2^n + 1$ for some $n \in \mathbb{N}$. Each block has a LOD level $0 \leq l \leq n$ that changes dynamically at runtime. Each representation of a block at a specific LOD level is called a *GeoMipMap*. For each GeoMipMap, the number of vertices on one side is $2^l + 1$ and the number of quads is 2^{2l}. Figure 3.2 shows an example of a 5×5 block at LOD levels 2, 1 and 0.



(a) LOD level 2 (maximum). (b) LOD level 1. (c) LOD level 0 (minimum).

Figure 3.2: Example of each GeoMipMap of a 5×5 block. The omitted vertices of lower LOD GeoMipMaps are marked as dotted circles (based on [dB00]).

The organisation of the terrain into blocks allows for easy view-frustum culling, which is performed with a quadtree, where each node contains the AABB of its four children and the leaf nodes contain the actual blocks.

The LOD level for each block is selected at runtime and is based on the screen-space error that is caused by changing the LOD level of a block. When the LOD level of a block changes, vertices get added or removed from the block, which causes a difference in height δ between the two GeoMipMaps of that block. Projecting δ into screen-space yields ε. This ε can be limited with a threshold τ, such that the change in LOD level occurs only if $\varepsilon < \tau$. The LOD selection can be sped up by pre-computing ε per GeoMipMap and storing it in a look-up table.

GeoMipMapping avoids cracks by checking the four neighboring blocks of a block and omitting the vertices that would cause cracks in the terrain. The

vertex omission is performed by rendering the bordering row/column of the current block as triangle fans, as shown in figure 3.3.

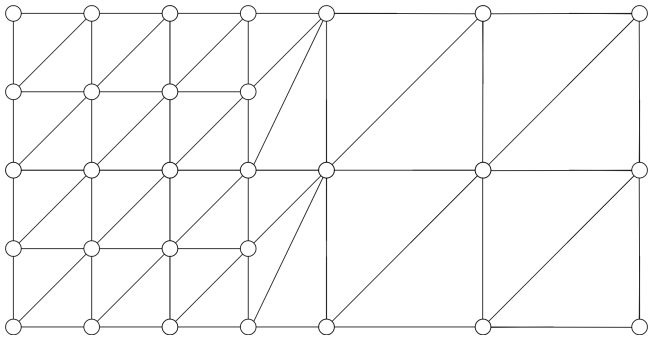


Figure 3.3: Example of GeoMipMapping's crack avoidance between a LOD 2 and a LOD 1 GeoMipMap of two 5×5 blocks (based on [dB00]).

Some further optimizations that were mentioned, which extend the just described basic GeoMipMapping algorithm, are *trilinear GeoMipMapping* (i.e. morphing the vertices at LOD transitions similarly to trilinear mipmapping), and *progressive GeoMipMap streaming*.

3.1.3 (GPU-based) Geometry Clipmaps

Geometry Clipmaps [LH04] is a terrain rendering technique published by Hoppe and Losasso in 2004. A follow-up GPU-based variant of Geometry Clipmaps [AH05] was published in GPU Gems 2 by Hoppe and Asirvatham in 2005. In this section, the basic features of the GPU-based Geometry Clipmaps algorithm are described, and we leave out some more advanced features, such as compression and noise-generated details.

The algorithm is based on a single flat mesh centered around the camera. The flat mesh is organized as a set of nested rings of l levels, where the innermost level $l-1$ is a filled-in $n \times n$ grid, and where the ring at level i is twice as big as the ring at level $i+1$. This n must be of the form $2^k - 1$ for some $k \in \mathbb{N}$. Each ring at a level is organized into 12 blocks of size $m \times m$, where $m = (n+1)/4$. Gaps inbetween the blocks are filled up with special types of blocks, namely the $m \times 3$ *ring fix-up* and the $(2m+1) \times n$ *interior trim*. In order to avoid T-junctions, a string of degenerate triangles is rendered at the border between blocks of different size. Since each block is identical up to translation and uniform scale, they get stored once on a vertex and index buffer and translated and scaled in the vertex shader at runtime, which greatly reduces memory consumption. Figure 3.4 shows an example of this mesh.

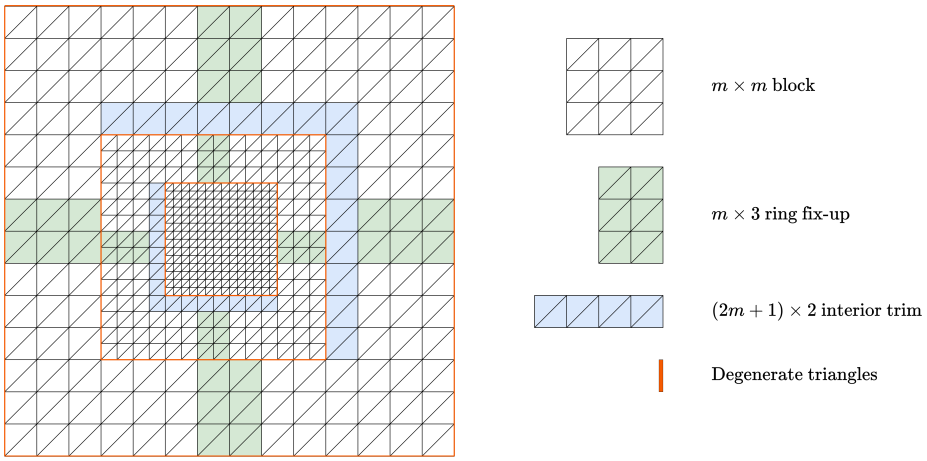


Figure 3.4: Example of the flat mesh in Geometry Clipmaps with $n = 15$, $m = 4$ and $l = 3$ (based on [AH05]).

In the vertex shader, the algorithm samples the height values from the heightmap texture. Additionally, it performs the calculations for the so-called *transition regions*, which are regions near the border of two levels in which the levels get morphed, so that the transition between levels is smooth and no popping occurs. The morphing is performed by computing the blend factor α, which is based on the position of the camera and the position of the vertex in world-space. This factor α is defined such that it is 0 everywhere except at the transition region, where it linearly grows from 0 to 1 until reaching the border.

During rendering, view-frustum culling is performed by intersecting each block with the view-frustum, and if the AABB of the block does not intersect the view-frustum, it does not get rendered.

Shading is performed with a normal map, which has twice the resolution of the heightmap.

3.1.4 Concurrent Binary Trees

The *concurrent binary tree (CBT)* [Dup20] is a data structure published by Dupuy in 2020. It essentially allows for binary trees to be computed in parallel using a binary heap represented as a bitfield. This allows for easy concurrent manipulation of tree nodes using bitwise operations. The data structure is applicable to problems relying on binary trees, such as the *longest edge bisection*.

The paper contains a section which describes the application of CBTs to terrain rendering. The approach is similar to [DWS+97] in the sense that it computes a triangulation of the terrain using bintree splitting and merging. The main difference is that the spliting and merging of the bintrees happen in parallel on compute shaders with the CBT data structure, whereas in [DWS+97], the bintrees are split and merged on the CPU.

The split and merge criteria for the triangles are defined such that sub-pixel

rasterization is avoided. Bintrees outside of the view-frustum and triangles at flat areas are not split any further, but no actual view-frustum culling gets performed.

An issue which is not adressed in the paper is how popping is avoided in the terrain.

3.1.5 Conclusion

In this subsection, the algorithms and their suitability for implementation are discussed.

ROAM ROAM is not particularly suited for today's GPU, since it mainly relies on immediate mode rendering [LH04], which is outdated in most graphics APIs of today. In addition to this, the costly splits and merges of the priority queues happen entirely on the CPU, which is undesirable, since this puts a heavy strain on the CPU.

GeoMipMapping The strong points of GeoMipMapping are the fact that its easy to understand and to implement. The algorithm was originally designed for immediate mode rendering in mind, which is as previously mentioned outdated nowadays. In order for GeoMipMapping to be suitable for modern GPUs, it needs to be modified so that it can work with vertex and index buffers, which is is feasible thanks to its block-based nature.

Geometry Clipmaps GPU-based Geometry Clipmaps was one of the first algorithms to utilize the vertex shader texture sampling functionality, which was a new feature of GPUs at the time. The fact that only very few vertices and indices are required on the GPU and the fact that the heightmap can be sampled in the vertex shader make GPU-based Geometry Clipmaps still a suitable algorithm for modern hardware. This is reflected in the fact that one of the most widely-used terrain plugins for Godot [Gil] is based on GPU-based Geometry Clipmaps (see the next section). Some other strong points of the algorithm are the transition regions for avoiding pops, its configurability, and the fact that no LOD determination needs to be performed, since the mesh is constant and the LOD is purely distance based.

CBT The CBT data structure "revitalized" mesh-subdivision-based approaches such as [LKR+96, DWS+97], since bintrees can now be computed in parallel with compute shaders, rather than sequentially on the CPU. It is a rather new approach that has yet to be tested in the real-world.

Overall Conclusion Overall, a suitable algorithm loads the vertices and indices once to the GPU and does not modify the buffers at runtime. Good candidates for this are GeoMipMapping with some modifications and GPU-based Geometry Clipmaps.

3.2 Terrain LOD in Real-world Systems

This section gives a short overview of real-world systems, such as game engines, which use terrain LOD algorithms.

3.2.1 Game Engines

Godot

Godot is a cross-platform game engine written in C#, C++ and its own scripting language GDScript. Terrains are supported in form of extensions developed by community members, which can be installed and used in Godot projects by game developers.

One such extension is *Terrain3D* by Cory Petkovsek [Pet] written in C++ for Godot 4. The LOD approach used in this extension is based on GPU-based Geometry Clipmaps by Hoppe and Losasso [AH05]. The concrete implementation of the mesh management is based on the Geometry Clipmaps implementation by Mike J Savage [Sav17].

Another extension for terrains is the *Godot Heightmap Plugin* by Marc Gilleron [Gil] written in GDScript and C++. The extension uses a quadtree-based approach for terrain LOD.

Unity

Unity is another cross-platform game engine written in C# and C++, and has a built-in terrain system. The core engine source code of Unity is only accessible by owning an enterprise licence, therefore no information is given on which specific terrain LOD algorithm is used for the built-in terrain in Unity. Instead, a high-level overview of Unity's terrain system and some additional information on related projects is given.

Unity's terrain system supports importing and exporting of heightmaps in the 8-bit or 16-bit grayscale RAW file format. The maximum heightmap size is 4097×4097, but the terrain is allowed to take dimensions larger than that. Visually, the mesh of the terrain LOD resembles that from a quadtree-based LOD approach, such as [Ulr02]. A pixel error value can be set, which determines how much the height of the LOD terrain can deviate from the actual height at that point. The Unity terrain does not perform any morphing between different LOD levels, which means that pops are visible at LOD level changes.

There exists an open-source library for hierarchical LOD in Unity called *HLODSystem* [Seo] developed by JangKyu Seo at Unity. HLODSystem also supports terrains with its TerrainHLOD component, allowing for conversion from an Unity Terrain object to a HLOD mesh with configurable parameters, such as chunk size and border vertex count. HLODSystem allows the developer to specify the mesh simplifier to be used and currently the only supported simplifier is *UnityMeshSimplifier* [Tec] that utilizes the *fast quadric mesh simplification* algorithm developed by Sven Forstmann [For].

The previously described CBT data structure and its application to terrain rendering was published by Dupuy at Unity Labs. At SIGGRAPH Courses 2021,

Deliot et al. gave a talk in which they described some additional implementation details and the (potential) integration into the Unity game engine [DDKY21]. As of today, it is unknown whether the CBT data structure was integrated into Unity's terrain system.

Unreal Engine

Unreal Engine is another cross-platform game engine written in C++ and features an integrated terrain system called the Landscape system. Most of the information described in this section is taken from the Unreal Engine 5.3 documentation [Gam].

The landscape system splits up the landscape into landscape components, similarly to [dB00], and stores its height data in texture images, similarly to [AH05] and [Str09]. The texture uses 32-bit per pixel, with 16-bits being reserved for the height, which is stored in the R and G channels, and 28-bit for the normals, stored in the B and A channels. Storing the normals on the GPU as well allows for high-resolution lighting and shading, regardless of the current LOD level. The maximum supported heightmap size is 8192×8192, which corresponds to the maximum supported texture size of many GPUs nowadays.

The texture mipmapping functionality is used for LOD, where each mipmap corresponds to a LOD. The transition between LOD levels gets morphed, as shown in figure 3.5.

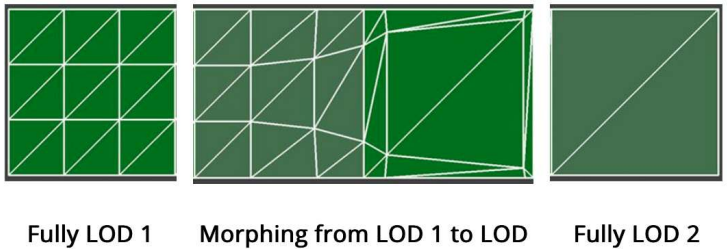


Figure 3.5: Unreal Engine's landscape system LOD morphing (taken from [Gam]).

Distant areas get streamed from and to the disk as the camera approaches or leaves them, respectively.

Frostbite

Frostbite is a closed-source game engine developed by DICE and is known for the *Battlefield* series. DICE has held numerous talks in the last few years describing iterations of their terrain system.

In 2007, Andersson at DICE published a paper describing the terrain rendering of the Frostbite game engine. Their approach stores a flat mesh of size 33×33 in a single vertex buffer, which gets and translated to its actual position at runtime. The heightmap is stored in a texture image and sampled in the vertex shader, similarly to [AH05]. The view-frustum culling is performed with a quadtree

and cracks in the terrain get avoided in a similar way to [dB00], except that crack-causing vertices get removed in the higher resolution mesh, rather than in the lower resolution mesh. This means that only 9 permutations of the mesh need to be stored, rather than 16.

During the Game Developers Conference 2012, Widmark DICE presented the terrain system of *Battlefield 3*, which was developed with their Frostbite 2 engine [Wid12]. The system is a quadtree-based terrain LOD system and improves on their previous work with a greater focus on paging and streaming of terrain data.

Chapter 4

ATLOD: A Terrain Level of Detail (Renderer)

This chapter describes *ATLOD* (short for **A** **T**errain **L**evel **o**f **D**etail (Renderer)), the demo terrain rendering application. The implemented algorithm is mainly based on GeoMipMapping, but also draws some inspiration from GPU-based Geometry Clipmaps and other algorithms, notably in its effective usage of the GPU.

4.1 Used Technologies

ATLOD is written in C++17 and OpenGL 4.2. For compiling build files, CMake (minimum version 3.5) is used. ATLOD uses the following third-party libraries:

- GLM: The *OpenGL Mathematics (GLM)* library provides functionality for the mathematics of graphics programming, such as classes for vectors, matrices and perspective transformations.
- GLEW: The *OpenGL Extension Wranger Library (GLEW)* is an extension loading library for OpenGL.
- GLFW: *GLFW* is a multi-platform library for desktop-based OpenGL applications, offering an API for managing windows, contexts and input handling.
- ImGui: *Dear ImGui* is a multi-platform graphical user interface library developed by Omar Cornut [Cor].
- STB: STB is a collection of header-only libraries developed by Sean Barrett [Bar]. ATLOD uses `stb_image.h` for loading images of heightmaps and textures.

ATLOD was developed with Qt Creator 9.6.1. The source code is hosted on GitHub on the repository AmarTabakovic/3d-terrain-with-lod and is licensed under the MIT license.

4.2 Basic Setup and Architecture

4.2.1 Overview

ATLOD consists of the following C++ source and header files:

- `main.cpp`: entry point of the application
- `application.cpp` and `application.h`: contains functions for the ImGui user interface and for rendering. All defined functions and global variables are inside the `Application` namespace.
- `terrain.cpp` and `terrain.h`: contains the abstract `Terrain` class (described in greater detail in the "Base Terrain" subsection).
- `heightmap.cpp` and `heightmap.h`: contains the `Heightmap` class (described in greater detail in the "Heightmap" subsection).
- `camera.cpp` and `camera.h`: contains the `Camera` class and the `Frustum` and `Plane` structs (described in greater detail in the "Camera" subsection).
- `shader.cpp` and `shader.h`: contains the `Shader` class (described in greater detail in the "Shaders" subsection).
- `skybox.cpp` and `skybox.h`: contains the `Skybox` class (described in greater detail in the "Skybox" subsection).

The algorithm implementations are stored specially in folders for them specifically. This decision was made in case multiple C++ source and header files were required for a single algorithm:

- `/naiverenderer`: contains the `naiverenderer.cpp` and `naiverenderer.h` files (described in greater detail in the "Naive Brute-force Algorithm" section).
- `/geomipmapping`: contains the `geomipmapping.cpp` and `geomipmapping.h` files (described in greater detail in the "GeoMipMapping" section).

GLSL files are stored in the folder `/glsl`.

Height data, overlay textures and skybox textures are stored in the `data` folder (outside the `src` folder), which needs to adhere to a specific structure in order for ATLOD to work properly. The details on how the folder needs to be structured are described in the `README` of the GitHub repository.

4.2.2 Command Line Arguments

ATLOD requires some command line arguments to be passed when starting. The exact required and optional command line arguments can be found in the `README` file of the GitHub repository.

The parsing of command line arguments happens in the function `parseArguments()` which takes in `argc` and `argv`, and is the first function called in `main()`. The arguments must be of the form `--arg_name=value` and no spaces are allowed in the arguments (e.g. file paths cannot contain spaces). If an argument is

invalid or a required argument was not passed, ATLOD simply prints the error message and exits. Otherwise, ATLOD continues with the initialization.

4.2.3 Shaders

The class `Shader` encapsulates an OpenGL shader program consisting of a vertex shader and a fragment shader. It is based on the "Shaders" chapter in *Learn OpenGL - Graphics Programming* [dV20]. The `Shader` class contains various methods to set uniform variables. The shader program is compiled in the constructor and can be used with the method `use()`.

4.2.4 Camera

The "Camera" class is based on the "Camera" chapter in *Learn OpenGL - Graphics Programming* [dV20]. It contains various fields which are usually found in virtual cameras, such as

- `Frustum _viewFrustum`: the view-frustum used for frustum culling (described in greater detail in the following subsection).
- `glm::vec3 _position`: the current position of the camera.
- `glm::vec3 _front`: the camera's front vector.
- `glm::vec3 _up`: the camera's up vector.
- `glm::vec3 _right`: the camera's right vector.
- `glm::vec3 _worldUp`: the camera's up vector in world-space.
- `float _zNear`: the z-coordinate of the near plane.
- `float _zFar`: the z-coordinate of the far plane.
- `float _aspectRatio`: the ratio of the window width and window height.
- `float _zoom`: the current FOV in degrees.
- `float _yaw`: the current yaw in degrees.
- `float _pitch`: the current pitch in the degrees.
- `float _movementSpeed`: the movement velocity.
- `float _lookSpeed`: the look-around velocity.

View-frustum Culling

The struct `Frustum` is defined with six fields of type `Plane` (one for each face), and the struct `Plane` is defined by the fields `glm::vec3 normal` and `float distance`, which correspond to the mathematical definitions of a frustum and a plane respectively.

The `Camera` class contains methods that check whether a given AABB intersects with the view-frustum. These methods are based on the chapter "Frustum Culling" in *Learn OpenGL - Graphics Programming* [dV20]. View-frustum culling is implemented in the methods `insideViewFrustum()` and `checkPlane()`. The method `insideViewFrustum()` takes two arguments `glm::vec3 p1` and

`glm::vec3 p2`, which correspond to the two points defining an AABB, and returns true if the AABB defined by these two points is inside the view-frustum, false otherwise.

Automatic Flying and Rotation

ATLOD supports automatic flying of the camera using two given world-space coordinates. The coordinates can be entered in a dialogue window and the flight velocity is adjustable with a slider.

The flying is implemented by linearly interpolating between the starting coordinate $\mathbf{p}_{start}$ and the end coordinate $\mathbf{p}_{end}$ with an interpolation factor t in the main game loop. More precisely, the direction is calculated by precomputing the direction vector $\mathbf{v}_{flyDir} = \mathbf{p}_{end} - \mathbf{p}_{start}$ and then by calculating

$$\mathbf{p}_{new} = \mathbf{p}_{start} + t \cdot \mathbf{v}_{flyDir}.$$

The interpolation factor t starts at 0 and gets increased by a small value $0 < t_{step} \leq 1$ every frame until $t = 1$. This t_{step} is adjustable by the user and corresponds to the previously mentioned flight velocity.

The class `Camera` contains a method `lerpFly()` which gets called each frame and performs the above calculation, as shown in listing **??**.

```
1 void Camera::lerpFly(float lerpFactor)
2 {
3     _position = origin + direction * lerpFactor;
4 }
```

The main render loop contains the snippet shown in listing **??**

```
1  float posLerp = 0.0f;
2  // ...
3  void run() {
4      while (!glfwWindowShouldClose(window)) {
5          // ...
6          if (camera.isFlying) {
7              camera.lerpFly(posLerp);
8              posLerp += 0.0005 + flightVel / 50000;
9
10             if (posLerp >= 1.0f) {
11                 camera.isFlying = false;
12                 posLerp = 0.0f;
13             }
14         }
15         // ...
16     }
17 }
```

The automatic camera rotation works similarly, but interpolates from the initial yaw yaw_{init} to $yaw_{init} + 2\pi$ with

$$yaw_{new} = yaw_{init} + t \cdot 2\pi.$$

4.2.5 Skybox

A *skybox* is a box in world space that simulates the sky using six texture images, one for each side of the box. The skybox implemented in ATLOD is based on the "Cubemaps" section in the "Advanced OpenGL" chapter in *Learn OpenGL - Graphics Programming* [dV20]. It is encapsulated in the class `Skybox` and contains the methods `loadTextures()`, `loadBuffers()`, `render()` and `unloadBuffers()`. Skyboxes are rendered as *cubemaps*.

For a skybox, each of its six texture images `front.png`, `back.png`, `left.png`, `right.png`, `top.png` and `bottom.png` is stored in the folder of that particular skybox, which is stored in the `data/skyboxes` folder. Figure 4.1 shows ATLOD's default skybox.

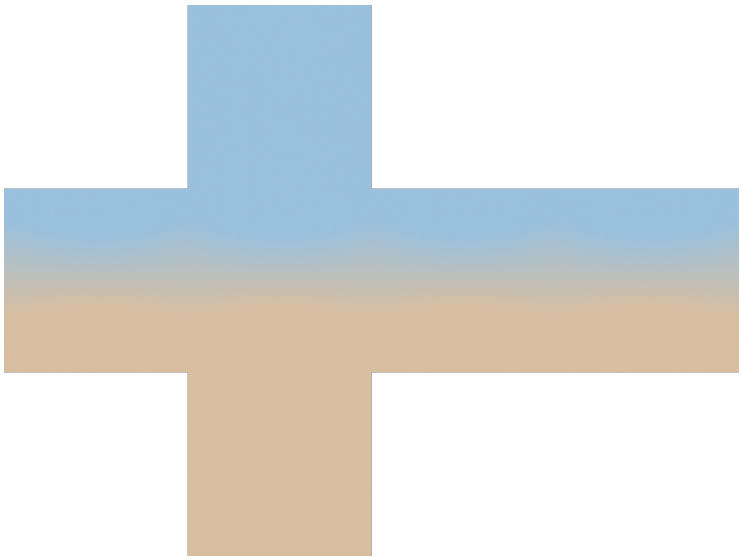


Figure 4.1: The default sunset gradient skybox.

An improvement over the current skybox system would be to actually calculate the atmospheric scattering, which would deliver a more realistic and flexible time-of-the-day-based lighting. A suitable approach would be *precomputed atmospheric scattering* by Bruneton and Neyret [BN08].

4.2.6 Heightmaps

The class `Heightmap` represents a heightmap and its data. Like many game engines today, such as Unity and Unreal Engine, ATLOD supports heightmaps as 16-bit grayscale PNG images, which allow for strorage of up to $2^{16}-1 = 65535$ height values per pixel. Unlike many game engines, heightmaps are not required to be square or a power of 2.

Heightmap Preprocessing

Digital elevation model (DEM) data is commonly offered in the GeoTIFF and Esri ASCII grid file formats by various DEM providers, such as SwissTopo and OpenTopography. These files can be converted into PNG using GIS software, such as QGIS or GDAL. Appendix A describes the process to convert a DEM from a GeoTIFF file or an Esri ASCII grid file into a 16-bit grayscale PNG image.

Loading

The method `load()` is responsible for loading a heightmap located at a given file path `fileName`. The height values are stored in the field `_data` of type `std::vector<unsigned short>`.

For terrain LOD algorithms using heightmap displacement inside the vertex shader, the `load()` method offers the possibility to optionally load the heightmap directly into an OpenGL texture object. The ID is stored on the current `Heightmap` instance in the field `_heightmapTextureId`, so that multiple different `Terrain` instances can share the same heightmap texture if needed.

4.2.7 Base Terrain

The base `Terrain` class is the superclass of all terrain LOD algorithms and contains fields that are common between different terrain LOD algorithms, namely the following:

- `Heightmap _heightmap`: the heightmap of the terrain.
- `Shader _shader`: the shader program for rendering the terrain.
- `unsigned _width, _heigth`: the width and height of the terrain. The reason for storing the width and height in the terrain as well is because the effective terrain dimensions can differ from the heightmap dimensions, as is the case in the GeoMipMapping of this implementation (described in more detail in the "GeoMipMapping" section).
- `unsigned _textureId`: the ID of the texture object for the overlay texture.
- `bool _hasTexture`: true if the method `loadTexture()` was called, false otherwise.
- `_xzScale`: the scaling variable in the xz-directions. This field is mostly unused in the implementation and simply set to 1 in most cases.
- `_yScale`: the scaling variable in the y-direction. This field is used everywhere where the y-coordinate of a vertex appears.

It also contains the three virtual methods `loadBuffers()`, `render()` and `unloadBuffers()`, which all terrain subclasses must implement. The `loadBuffers()` should be called after instantiating the terrain and before rendering. The method `render()` should be called every frame in the main rendering loop and the method `unloadBuffers()` should be called before destroying the current terrain instance.

The class also contains the method `loadTexture()` to load an overlay texture, which gets stored in the field `_textureId`;

4.3 Naive Brute-force Algorithm

The naive brute-force algorithm, which simply renders every vertex without any LOD considerations, is encapsulated in the class `NaiveRenderer`.

4.3.1 Vertex and Index Organisation

A vertex consists of its (x, y, z)-position, its normal vector (n_x, n_y, n_z) and of its texture coordinates (u, v). All components are 4-byte floating point values, which means that per vertex, $4 \times 8 = 32$ bytes of GPU memory get allocated. These attributes are organized in a vertex array object stored in the field `_vao`.

The indices are organized such that they can be rendered as triangle strips with `GL_TRIANGLE_STRIPS`. Each row is separated using a special marker index named `RESTART`, which is set to the maximum possible `GLuint` value and is used for the `GL_PRIMITIVE_RESTART` mode, allowing for the entire terrain to be rendered in a single `glDrawElements()` call. This draw call happens every frame in the method `render()`. Figure 4.2 shows the organization of indices for rendering the terrain as triangle strips.

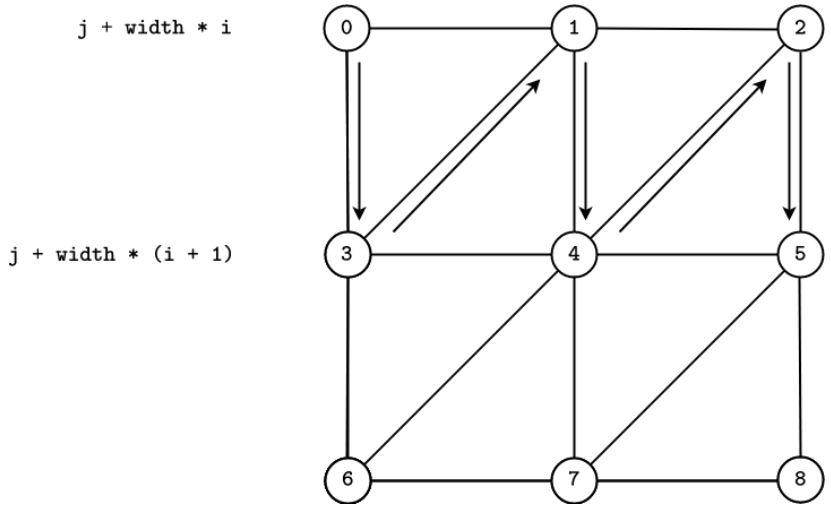


Figure 4.2: Example of a terrain layout for triangle strips. The looping index `i` goes from 0 to the terrain height and `j` from 0 to the terrain width. The final indices to be rendered are 0, 3, 1, 4, 2, 5, `RESTART`, 3, 6, 4, 7, 5, 8, `RESTART`.

Loading

The method `loadBuffers()` is responsible for loading the data into the vertex and index buffer. The first step is the generation of the normal vectors. This

is done in the method `loadNormals()`, where the normals of each vertex are calculated into an intermediate vector `_normals`.

The normals of the vertices are calculated by summing up the normal vectors of the four adjacent faces, which are given by calculating the cross product of the adjacent edges. The adjacent edges are given by subtracting neighboring vertices, as shown in figure 4.3.

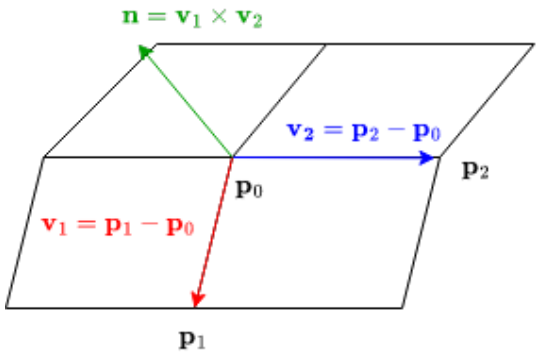


Figure 4.3: Example of a normal vector calculation of the bottom right face. The same process gets repeated for the other three adjacent faces.

Afterwards the generation of the normal vectors, the vertices get generated and loaded onto the vertex buffer. The same happens for the indices. After the data is loaded onto the GPU, the intermediate vectors `_normals`, `_vertices` and `_indices` are cleared so that no unnecessary memory waste occurs.

4.3.2 Rendering

Draw Call

The entire terrain gets rendered using 1 draw call, thanks to the index organisation with triangle strips and primitive restarting. Listing

Vertex Shader

The vertex shader simply applies the model, view and projection matrices to the current position attribute, which is set as the fragment position output variable. The texture coordinate output variable is set to the vertex attribute.

The normal vectors require a small additional operation before being sent to the fragment shader: in order to allow for lighting with non-uniform scaling with the model matrix, the *normal matrix* must be applied to the normal vectors first [dV20]. The normal matrix is computed *before* being set as a uniform variable, as follows: $\mathbf{N} = (\mathbf{M}^{-1})^{\top}$. Afterwards, it is converted to a 3×3 matrix in the vertex shader and multiplied with the normal vector, which is then set as the normal output variable.

Fragment Shader

In the fragment shader, the lighting is computed with Phong shading and the overlay texture is applied (if one is used). The lighting consists of an ambient light with a strength factor 0.5 and of a diffuse light. Both lights have a white color.

The fog calculation is based on [Sal15] and can be adjusted with the argument `density`. Listing 4.1 shows the function which calculates the fog factor and its usage:

```
1  float calculateFog(float density) {
2    float dist = length(cameraPos - FragPosition);
3    float fogFactor = exp(-density * dist);
4    return clamp(fogFactor, 0.0f, 1.0f);
5  }
6
7  void main() {
8    // ...
9    float fogFactor = calculateFog(fogDensity);
10   color = mix(fogColour, color, fogFactor);
11   // ...
12   FragColor = vec4(color, 1.0f);
13 }
```

Listing 4.1: The fog calculation in the fragment shader.

4.4 GeoMipMapping

This implementation of GeoMipMapping supports most basic functionalities described in the original paper, but differs in a few key aspects: it draws some inspiration from other approaches, most notably from GPU-based Geometry Clipmaps [AH05] and certain minor elements from "Terrain Rendering in Frostbite Using Procedural Shader Splatting" [And07].

Both approaches utilize a single flat mesh (positioned around the viewer in [AH05], in world-space [And07]) and store the heightmap as a texture object. The height values are sampled in the vertex shader, which are then used to displace the flat mesh on the y-axis.

The idea of using a texture image for the heightmap is applied to ATLOD's GeoMipMapping implementation. Rather than generating vertex buffers for each block and loading in the height values into the vertices directly (as in the naive renderer implementation), a single flat mesh with the side length of the block size is generated once at load time. At render time, for each block, the mesh is translated to the block's world-space position and the height values get sampled from the heightmap stored on the GPU. The upcoming subsections describe the approach in greater detail.

4.4.1 Class Structure

GeoMipMapping contains the following members:

- `unsigned _blockSize`: the size of a single block.
- `unsigned _nBlocksX, _nBlocksZ`: the number of blocks in the x and z-direction. These two values are calculated in the constructor by dividing the terrain width and height by the block size and rounding the values down.
- `std::vector<GeoMipMappingBlock> _blocks`: stores the blocks.
- `std::vector<unsigned> _borderSizes, _borderStarts, _centerSizes, _centerStarts`: Stores the start indices and sizes of the subsets of the index buffer containing the border areas and center areas (explained in more detail subsection "Vertex and Index Organisation").
- `unsigned _vao, _vbo, _ebo`: the vertex array object, vertex buffer object and element buffer object respectively.
- `float _baseDistance`: the distance until the next lower LOD level is chosen (explained in more detail in subsection "LOD Selection").
- `unsigned _minLod, _maxLod, _maxPossibleLod`: the minimum and maximum user set LOD level, and the maximum possible LOD level.

4.4.2 Blocks

As previously described in the high-level overview of the GeoMipMapping algorithm, the algorithm splits up the terrain into square blocks of side length $2^n + 1$.

In this implementation, a block is simply a structure containing information for a particular section of the terrain. The struct `GeoMipMappingBlock` represents such a block and contains the following fields:

- `unsigned blockId`: the ID of the current block.
- `unsigned currentBorderBitmap`: the current border permutation as a bitmap.
- `glm::vec2 translation`: the 2-dimensional translation vector for translating the flat mesh to the block's actual world-space position.
- `glm::vec3 worldCenter`: the actual center position of that block in world-space, i.e. its y-coordinate is computed from the heightmap. This field is used for the distance calculation for the LOD determination during rendering.
- `glm::vec3 p1, p2`: the two points defining the AABB of the block (corresponding to $\mathbf{p}_1$ and $\mathbf{p}_2$ in the subsection "View-frustum Culling" in "Basics of Terrain Rendering").

4.4.3 Vertex and Index Organisation

ATLOD's GeoMipMapping implementation consists of a single vertex buffer and index buffer. The vertex buffer contains the vertices for a single flat $blockSize \times blockSize$ mesh centered around $(0, 0, 0)$, where $blockSize = 2^n + 1$ and n is the maximum LOD level. The vertices of the flat mesh only consist of 4-byte floating point (x, z)-coordinates, since the mesh is flat.

The index buffer contains the 4-byte unsigned integer indices of the flat mesh and is organized as follows: the flat mesh is split up into its border area and center area. The reason for splitting the mesh up this way will be made clear shortly. The first part of the index buffer stores the indices of the border area for every LOD level and border permutation, and the second part of the index buffer stores the center area for every LOD level. What a *border permutation* is will be explained in the next section. Figure 4.4 shows the described index buffer organisation.

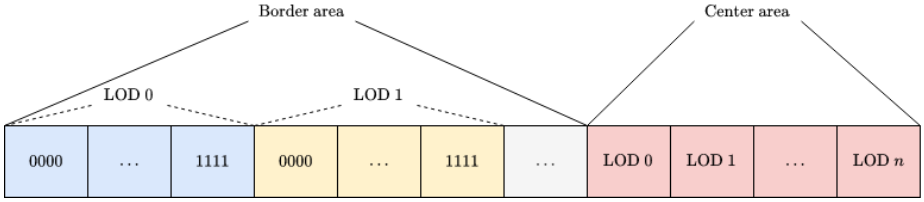
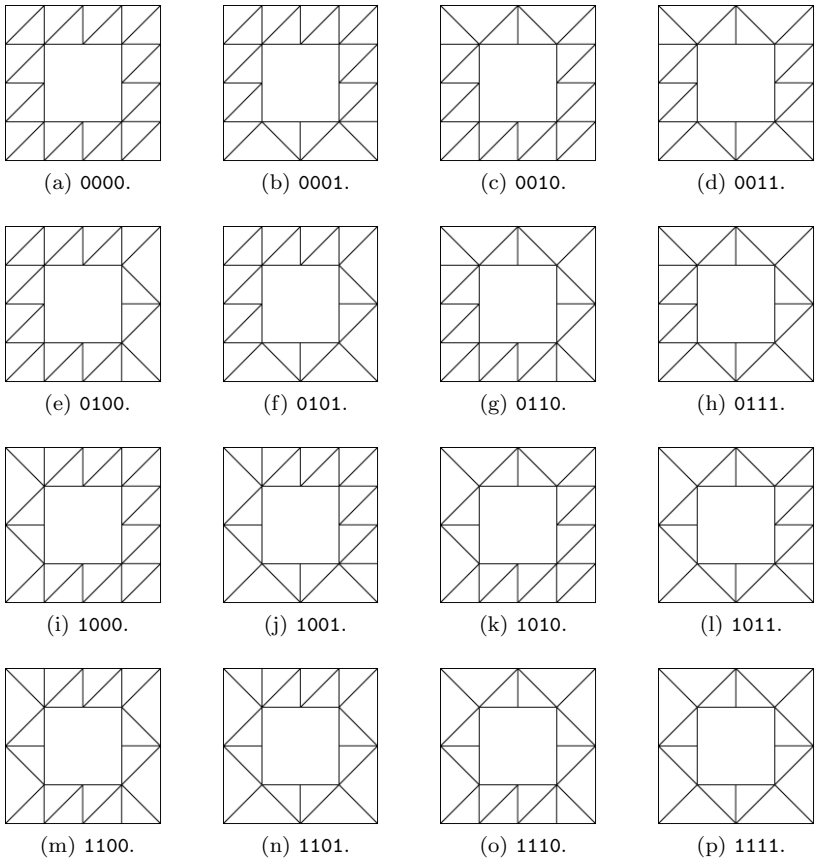


Figure 4.4: The index buffer organisation of the single flat block. The variable n corresponds to the maximum LOD level.

Border Permutations

A border permutation is defined to be a 4-tuple (t, b, l, r), where t, b, l, r correspond to top, bottom, left and right, and where each entry is set to 1 if the block on the corresponding side has a lower LOD, and 0 otherwise. For example, if the top and right neighboring blocks have a lower LOD than the current block, the border permutation is $(1, 0, 0, 1)$. These border permutations can also be expressed as bitfields, e.g. `1001`, which allows for easy indexing into the subset of the index buffer containing the relevant indices. The number of possible permutations is $2^4 = 16$. In order for this approach to work, the difference in LOD level between any two bordering blocks must be at most 1. Figure 4.5 shows all possible border permutations of a 5×5 block at LOD level 2.



(a) 0000. (b) 0001. (c) 0010. (d) 0011.

(e) 0100. (f) 0101. (g) 0110. (h) 0111.

(i) 1000. (j) 1001. (k) 1010. (l) 1011.

(m) 1100. (n) 1101. (o) 1110. (p) 1111.

Figure 4.5: Every possible border permutation for a LOD 2 GeoMipMap of a 5×5 block. The center subblocks have been omitted from the illustration.

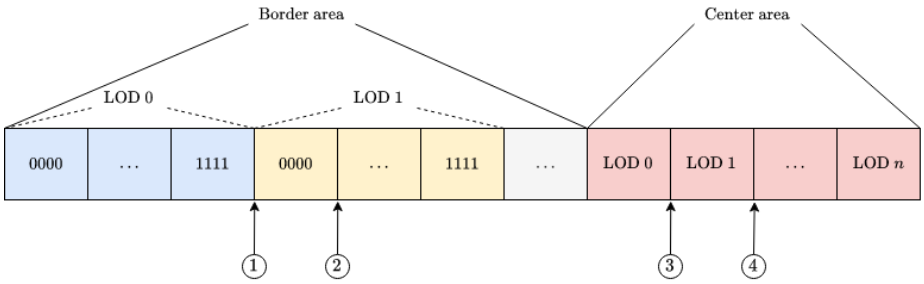
This makes clear why the flat mesh is split into its border and center area. The center area only depends on the LOD level and is the same regardless of the current border permutation. Not splitting the flat mesh up into its border and center area would require longer index buffer generation times and consume significantly more GPU memory.

Starts and Sizes Lists

The organisation of the index buffer as presented requires some additional management of the start indices and sizes of the subsets of the index buffer. As previously mentioned, the `GeoMipMapping` class contains four members of the type `std::vector<unsigned>`: `_borderStarts`, `_borderSizes`, `_centerStarts` and `_centerSizes`. The `_borderStarts` and `_borderSizes` lists store the starting index (into the index buffer) and the number of indices, for a subset of the index buffer containing the indices of the border area for a given border permutation and LOD level. Both the lists are indexed by multiplying the current LOD level by 16 and then adding the current border permutation to it. The `_centerStarts` and `_centerSizes` lists are indexed similarly, except that they

are indexed simply with the current LOD level.

In order to illustrate the idea more clearly, the following example is given: say that the current block has a LOD level of 1 and every neighboring block has the same LOD level (i.e. the border permutation is $(0, 0, 0, 0)$). We want to render the block, which means we need to retrieve the indices for the flat mesh at the LOD level 1 and for the border permutation `0000`. The start index and the size for the border area are retrieved with `_borderStarts[1 * 16 + 0b0000]` and `_borderSizes[1 * 16 + 0b0000]` respectively, and for the center area with `_centerStarts[1]` and `_centerSizes[1]`. These four values are passed to the two `glDrawElements()` draw calls for the block, which renders the flat mesh at the chosen LOD level 1 and border permutation $(0, 0, 0, 0)$. The full rendering process is described in the subsection "Rendering" of this section. Figure 4.6 illustrates this example.



① `_borderStarts[16 * 1 + 0b0000]`

② `_borderStarts[16 * 1 + 0b0000] + _borderSizes[16 * 1 + 0b0000]`

③ `_centerStarts[1]`

④ `_centerStarts[1] + _centerSizes[1]`

Figure 4.6: Illustration of accessing the start index and size of the subsets of the index buffer for LOD 1 and border permutation $(0, 0, 0, 0)$

Loading

Now that the organisation of the vertices and indices has been presented, the loading mechanisms are described. The vertex and index buffers get loaded in the method `loadBuffers()`, which calls the two helper methods `loadVertices()` an `loadIndices()`.

The method `loadVertices()` (listing 4.2) simply generates the vertex array object with its ID stored in the field `_vao` and loads the vertices of the flat mesh of size $blockSize \times blockSize$ centered around $(0, 0, 0)$ into a vertex buffer with its ID stored in the field `_vbo`.

```
1 void GeoMipMapping::loadVertices()
```

```
{
    for (int i = 0; i < _blockSize; i++) {
        for (int j = 0; j < _blockSize; j++) {
            // Load vertices around center point
            float x = (-(float)_blockSize / 2.0f + (float)
    _blockSize * j / (float)_blockSize);
            float z = (-(float)_blockSize / 2.0f + (float)
    _blockSize * i / (float)_blockSize);

            _vertices.push_back(x); // Position x
            _vertices.push_back(z); // Position z
        }
    }

    glGenVertexArrays(1, &_vao);
    glBindVertexArray(_vao);

    glGenBuffers(1, &_vbo);
    glBindBuffer(GL_ARRAY_BUFFER, _vbo);
    glBufferData(GL_ARRAY_BUFFER, _vertices.size() * sizeof(
    float), &_vertices[0], GL_STATIC_DRAW);

    // Position attribute
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 *
    sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
}
```

Listing 4.2: Method `GeoMipMapping::loadVertices()` that generates the vertex array object and loads the vertex buffer with the flat mesh of size $blockSize \times blockSize$ centered around $(0, 0, 0)$.

The index loading mechanism is significantly more complex. The top-level index loading method `loadIndices()` performs the following operations:

- It loads the LOD 0 and LOD 1 representation of the flat mesh using `loadLod0()` and `loadLod1()` respectively. The LOD 0 and LOD 1 indices require special treatment. They are stored only as borders and do not have a center. If `_minLod` is greater than 1, then this step is skipped.
- It loads the rest of the indices from the `_minLod` to `_maxLod`, by first loading in the border indices with `loadBorderAreaForLod()` and then the center areas with `loadCenterAreaForLod()`.

The `loadBorderAreaForLod()` method accepts the LOD level to be loaded `lod` and first iterates from 0 to 16 (i.e. for every possible border permutation) and calls `loadBorderAreaForPermutation(lod, i)`. The function `loadBorderAreaForPermutation()` takes the LOD level to be loaded and the border permutation to be loaded and calls the following 8 helper methods:

- `loadTopLeftCorner(step, permutation)`
- `loadTopBorder(step, permutation)`

- `loadTopRightCorner(step, permutation)`
- `loadRightBorder(step, permutation)`
- `loadBottomRightCorner(step, permutation)`
- `loadBottomBorder(step, permutation)`
- `loadBottomLeftCorner(step, permutation)`
- `loadLeftBorder(step, permutation)`

Each of the eight method calls loads in the corresponding side or corner at the LOD level given by `step` and for the border permutation given by `permutation`. The variable `step` is the step size (i.e. how many vertices to skip due to the LOD when computing the mesh) and is computed directly from `lod` with `step = std::pow(2, _maxLod - lod)`.

Listing 4.3 shows the method `loadTopLeftCorner()`. The other methods work similarly.

```
1  unsigned GeoMipMapping::loadTopLeftCorner(unsigned step,
       unsigned permutation)
2  {
3      unsigned count = 0;
4      if ((permutation & LEFT_BORDER_BITMASK) && (permutation
       & TOP_BORDER_BITMASK)) { /* bitmask is 1_1_ */
5          /*
6            * *- - -*- - -*
7            * |\          /|
8            * |  \      /  |
9            * |    \ /    |
10           * *     *- - -*
11           * |    /|
12           * |  /  |
13           * |/    |
14           * *- - -*
15           *
16           */
17
18         pushIndex(2 * step, step);
19         pushIndex(2 * step, 0);
20         pushIndex(step, step);
21         pushIndex(0, 0);
22         pushIndex(0, 2 * step);
23         indices.push_back(RESTART_INDEX);
24
25         pushIndex(step, 2 * step);
26         pushIndex(step, step);
27         pushIndex(0, 2 * step);
28         indices.push_back(RESTART_INDEX);
29
30         count += 10;
31
```

```
32      } else if (permutation & LEFT_BORDER_BITMASK) { /*
        bitmask is 1_0_*/
33
34          /*
35            * *- - -*- - -*
36            * |\     |     /|
37            * |  \   |   /  |
38            * |     \|/     |
39            * *      *- - -*
40            * |     /|
41            * |  /   |
42            * |/     |
43            * *- - -*
44            *
45            */
46
47          pushIndex(step, 0);
48          pushIndex(0, 0);
49          pushIndex(step, step);
50          pushIndex(0, 2 * step);
51          pushIndex(step, 2 * step);
52          indices.push_back(RESTART_INDEX);
53
54          pushIndex(step, 0);
55          pushIndex(step, step);
56          pushIndex(2 * step, 0);
57          pushIndex(2 * step, step);
58          indices.push_back(RESTART_INDEX);
59
60          count += 11;
61
62      } else if (permutation & TOP_BORDER_BITMASK) { /*
        bitmask is 0_1_ */
63          /*
64            * *- - -*- - -*
65            * |\          /|
66            * |  \      /  |
67            * |     \ /     |
68            * *- - -*- - -*
69            * |     /|
70            * |  /   |
71            * |/     |
72            * *- - -*
73            *
74            */
75
76          pushIndex(0, step);
77          pushIndex(0, 2 * step);
78          pushIndex(step, step);
79          pushIndex(step, 2 * step);
80          indices.push_back(RESTART_INDEX);
81
82          pushIndex(2 * step, step);
83          pushIndex(2 * step, 0);
```

```
84          pushIndex(step, step);
85          pushIndex(0, 0);
86          pushIndex(0, step);
87          indices.push_back(RESTART_INDEX);
88
89          count += 11;
90
91      } else { /* bitmask is 0_0_*/
92          /*
93            * *- - -*- - -*
94            * |    /|    /|
95            * |  /  |  /  |
96            * |/    |/    |
97            * *- - -*- - -*
98            * |    /|
99            * |  /  |
100           * |/    |
101           * *- - -*
102           *
103           */
104
105         pushIndex(0, step);
106         pushIndex(0, 2 * step);
107         pushIndex(step, step);
108         pushIndex(step, 2 * step);
109         indices.push_back(RESTART_INDEX);
110
111         pushIndex(0, 0);
112         pushIndex(0, step);
113         pushIndex(step, 0);
114         pushIndex(step, step);
115         pushIndex(2 * step, 0);
116         pushIndex(2 * step, step);
117         indices.push_back(RESTART_INDEX);
118
119         count += 12;
120     }
121
122     return count;
123 }
```

Listing 4.3: Method `GeoMipMapping::loadTopLeftCorner()` which loads in the indices of the top left corner of the flat mesh for a given LOD level and border permutation

The `loadCenterAreaForLod()` method accepts the LOD level to be loaded `lod` as well and simply loads the indices with the given LOD resolution into `_indices`, similarly to the index loading of the naive brute-force algorithm.

After loading the indices and the vertices, the intermediate vectors `_vertices` and `_indices` get cleared.

4.4.4 Rendering

The method `render()` that is called each frame performs in two phases:

- The first phase iterates through every block and updates the LOD level of the block (described in the next subsection "LOD Selection") and then updates the border permutation bitmap.
- The second phase iterates through every block again and performs view-frustum culling, sets the uniform variables in the shader, and finally renders the the border area and center area of that block.

LOD Selection

The LOD of ATLOD's GeoMipMapping implementation is based on the Euclidean distance *dist* between the camera's position $\mathbf{p}_{camPos}$ and the block's center point $\mathbf{p}_{blockCenter}$

$$dist = \sqrt{\begin{matrix}(blockCenter_x - camPos_x)^2 + (blockCenter_y - camPos_y)^2 \\ + (blockCenter_z - camPos_z)^2\end{matrix}}.$$

A minor optimization for this calculation is possible by instead calculating the LOD using the squared distance, which avoids an expensive square-root call.

Two different LOD determination modes are possible: the *linearly growing distance* and the *exponentially growing distance*. Both use a base distance value *baseDist* which can be set by the user. Note that *baseDist* should be larger than *blockSize*, as otherwise cracks in the terrain may occur, but also not too large, so that the performance is still adequate. The LOD computed by the linearly growing distance can be defined with the following recursive formula

$$lod_{lin}(dist, i) = \begin{cases} l - i + 1 & dist \leq i \cdot baseDist \\ lod_{lin}(dist, i+1) & i \cdot baseDist < dist < (l+1) \cdot baseDist \\ 0 & \text{otherwise} \end{cases},$$

where i starts at 1 and l is the maximum LOD level. As a basic example, say that $l = 3, baseDist = 100, dist = 250$. We begin computing the LOD level with $lod_{lin}(250, 1)$. The second condition $1 \cdot 100 < 250 < 4 \cdot 100$ holds, so we continue with $lod_{lin}(250, 2)$. Again, the second condition $2 \cdot 100 < 250 < 4 \cdot 100$ holds, so we continue with $lod_{lin}(250, 3)$. Now, the first condition $250 \leq 3 \cdot 100$ holds, so the entire expression evaluates to $3 - 3 + 1 = 1$, which means we set the LOD level of the block to 1.

The exponentially growing distance is defined very similarly:

$$lod_{exp}(dist, i) = \begin{cases} l - i + 1 & dist \leq i \cdot baseDist \\ lod_{exp}(dist, 2i) & i \cdot baseDist < dist < (l+1) \cdot baseDist \\ 0 & \text{otherwise} \end{cases}.$$

The above expressions are implemented in the `GeoMipMapping` class in a single method `determineLodDistance()`. The LOD determination mode can be

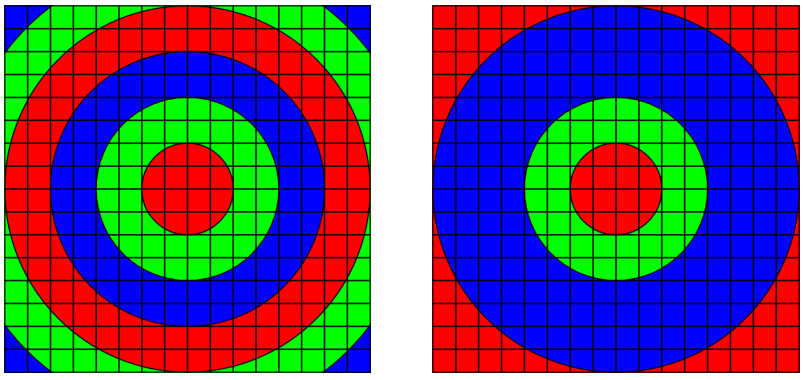
set with the boolean argument `doubleEachLevel`. Additionally, the minimum and maximum LOD levels can be manually set to be different from 0 and l respectively by the user, with the fields `_minLod` and `_maxLod`. Note that the user defined `_minLod` and `_maxLod` values are both set to $\max\{0, \texttt{\_minLod}\}$ and $\min\{l, \texttt{\_maxLod}\}$ respectively in the constructor, so that the LOD level does not go out of bounds. Listing 4.4 shows the method `determineLodDistance`.

```
unsigned GeoMipMapping::determineLodDistance(float
    squareDistance, float baseDist, bool doubleEachLevel)
{
    unsigned distancePower = 1;
    for (unsigned i = 0; i < _maxLod - _minLod; i++) {
        if (squaredDistance < distancePower * distancePower
    * baseDist * baseDist)
            return _maxLod - i;

        if (doubleEachLevel)
            distancePower <<= 1;
        else
            distancePower++;
    }
    return _minLod;
}
```

Listing 4.4: Method `GeoMipMapping::determineLodDistance()` that determines the LOD level of a block based on its distance to the camera.

Figure 4.7 shows both LOD determination modes in action.



(a) Linearly growing distance mode.

(b) Exponentially growing distance mode.

Figure 4.7: Illustration of a flat terrain showcasing the linearly growing distance mode (a) and exponentially growing distance mode (b). The red, green and blue colors indicate successively lower LOD levels, starting from the maximum level in the center.

Border Bitmap Calculation

Listing 4.5 shows the calculation of the border permutation bitmap for a given block. The first step consists of calculating the minimum and maximum x and z block indices in order to avoid going out of bounds. Afterwards, the LOD of the four neighboring blocks gets retrieved and the bitmap is set depending on whether the corresponding side has a lower LOD or not, as described previously.

```
1  unsigned GeoMipMapping::calculateBorderBitmap(unsigned
       currentBlockId, unsigned x, unsigned z)
2  {
3      unsigned currentLod = _blocks[currentBlockId].currentLod
       ;
4
5      unsigned maxX = std::max((int)x - 1, 0);
6      unsigned minX = std::min((int)x + 1, (int)_nBlocksX - 1)
       ;
7      unsigned maxZ = std::max((int)z - 1, 0);
8      unsigned minZ = std::min((int)z + 1, (int)_nBlocksZ - 1)
       ;
9
10     GeoMipMappingBlock& leftBlock = getBlock(maxX, z);
11     GeoMipMappingBlock& rightBlock = getBlock(minX, z);
12     GeoMipMappingBlock& topBlock = getBlock(x, maxZ);
13     GeoMipMappingBlock& bottomBlock = getBlock(x, minZ);
14
15     unsigned leftLower = currentLod > leftBlock.currentLod ?
        1 : 0;
16     unsigned rightLower = currentLod > rightBlock.currentLod
        ? 1 : 0;
17     unsigned topLower = currentLod > topBlock.currentLod ? 1
        : 0;
18     unsigned bottomLower = currentLod > bottomBlock.
       currentLod ? 1 : 0;
19
20     return (leftLower << 3) | (rightLower << 2) | (topLower
       << 1) | bottomLower;
21 }
```

Listing 4.5: The method `GeoMipMapping::calculateBorderBitmap()` which computes the border permutation bitmap for a given block.

Draw Calls

Two draw calls are performed per block: one for the center area and one for the border area. The arguments of the draw calls follow the logic described in "Vertex and Index Organisation". Listing 4.6 shows the two draw calls which occur inside the second phase of the `render()` method.

```
unsigned currentIndex = block.currentLod - _minLod;

// First render the center subblocks (only for LOD >= 2, since
// LOD 0 and 1 do not have a center block) */
if (block._currentLod >= 2) {
    glDrawElements(GL_TRIANGLE_STRIP,
        centerSizes[currentIndex],
        GL_UNSIGNED_INT,
        (void*)(centerStarts[currentIndex] * sizeof(unsigned)));
}

// Then render the border subblocks
currentIndex = currentIndex * 16 + block._currentBorderBitmap;
glDrawElements(GL_TRIANGLE_STRIP,
    borderSizes[currentIndex],
    GL_UNSIGNED_INT,
    (void*)(borderStarts[currentIndex] * sizeof(unsigned)));
```

Listing 4.6: The two draw calls occuring inside the second loop over all blocks inside the method `GeoMipMapping::render()`.

Vertex Shader

The vertex shader calculates the heightmap texture coordinate using the vertex position attribute, the texture width and height given as uniforms, and the block's translation vector given as an uniform.

Afterwards, it samples the height from the heightmap texture and multiplies it by 65535 (since the texture image is normalized with all values from 0 to 1).

Next, it translates the vertex from its initial position around $(0, 0, 0)$ to its actual world-space position using the translation vector and the y-coordinate is set to the sampled height. Listing 4.7 shows the source code of the vertex shader.

```
#version 330 core
layout (location = 0) in vec2 aPos;

out vec3 FragPosition;

uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;
uniform vec2 offset;
uniform sampler2D heightmapTexture;
uniform float textureWidth;
uniform float textureHeight;
```

```
13
14 void main()
15 {
16     vec2 texPos =  vec2((aPos.x + offset.x + 0.5 *
       textureWidth) / (textureWidth),
17                         (aPos.y + offset.y + 0.5 *
       textureHeight) / (textureHeight));
18
19     float height = texture(heightmapTexture, texPos).r;
20     float y = height * 65535;
21
22     vec3 actualPos = vec3(aPos.x + offset.x, y, aPos.y +
       offset.y);
23
24     FragPosition = vec3(model * vec4(actualPos, 1.0));
25     gl_Position = projection * view * model * vec4(actualPos
       , 1.0);
26 }
```

Listing 4.7: The vertex shader of the GeoMipMapping implementation.

Fragment Shader

The fragment shader shades the fragment using the Phong shading method and is exactly the same as the naive brute-force algorithm in terms of computing the lighting. The main difference is the way how normals are handled. Recall that the vertices contained no normal vector attribute. The normal vectors are instead calculated based on the method described in [And07]: first, the heightmap is sampled at the four orthogonally neighboring points for the height values $y_{left}, y_{right}, y_{top}, y_{bottom}$. Using these four values, the slope in x and z-direction can be calculated by computing

$$dx = y_{left} - y_{right}$$
$$dz = y_{top} - y_{bottom}.$$

These values can now be used to create a normal vector

$$\mathbf{n} = \frac{(dx, 2, dz)}{\|(dx, 2, dz)\|}.$$

Afterwards, this normal vector can be used as usual to compute diffuse lighting. Listing 4.8 shows the calculation of the normal vector from the heightmap texture.

```
1 vec2 texPos = vec2((FragPosition.x + 0.5 * textureWidth) /
      textureWidth, (FragPosition.z + 0.5 * textureHeight) /
      textureHeight);
2
3 float leftHeight = texture(heightmapTexture, texPos - vec2
      (1.0 / textureWidth, 0)).r;
```

```
4 float rightHeight = texture(heightmapTexture, texPos + vec2
      (1.0 / textureWidth, 0)).r;
5 float upHeight = texture(heightmapTexture, texPos + vec2(0,
      1.0 / textureHeight)).r;
6 float downHeight = texture(heightmapTexture, texPos - vec2
      (0, 1.0 / textureHeight)).r;
7
8 // Multiply with 65535 to denormalize
9 float dx = (leftHeight - rightHeight) * yScale * 65535;
10 float dz = (downHeight - upHeight) * yScale * 65535;
11
12 vec3 normal = normalize(vec3(dx, 2.0f, dz));
13
14 // Continue with diffuse lighting ...
```

Listing 4.8: Calculating the normal vector in the fragment shader using the heightmap texture.

The fog calculation is exactly the same as in the naive brute-force rendering algorithm (listing 4.1).

Chapter 5

Results

In this chapter, the performance and visual accuracy of ATLOD is measured.

5.1 Experimental Setup

5.1.1 Hardware

The hardware used is a MacBook Air 2020 with an Intel CPU. The specifications are displayed in table 5.1.

CPU	1.1 GHz Dual-Core Intel Core i3
Memory	8 GB 3733 MHz LPDDR4X
Graphics	Intel Iris Plus Graphics 1536 MB
OS	macOS Monterey Version 12.6
Resolution	2560×1600

Table 5.1: The specifications of the used MacBook Air 2020.

5.1.2 Height Data and GeoMipMapping Configuration

The height data used is the SRTM 30m data set retrieved from OpenTopography [NAS13]. The heightmap file is a 13922×14140 16-bit greyscale PNG image converted from a GeoTIFF file (figure 5.1) and covers a large extent of Switzerland (excluding the Grisons) and small parts of Germany, France and Italy. The total area is 130 km^2.

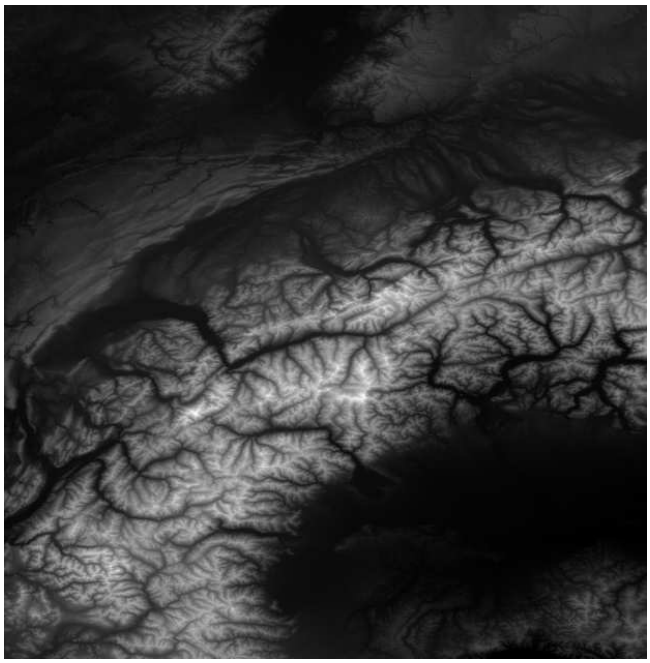


Figure 5.1: The 13922×14140 16-bit greyscale heightmap used for benchmarking (retrieved from OpenTopography [NAS13]). In this figure, the gray values were converted from 0,...,65535 to 0,...,255 in order to make the heights more visible.

The GeoMipMapping algorithm is configured for the best balance between performance and visual accuracy as shown in table 5.2.

Block size	$2^9 + 1 = 513$
Fog factor	0 (deactivated)
Minimum LOD	0 (default)
Maximum LOD	9 (default)
Base distance	700
LOD determination mode	Exponentially growing distance
y-scale	0.03

Table 5.2: Rendering settings for the benchmarks.

5.1.3 Benchmarks

Two kinds of benchmarks are performed: The *performance benchmark*, which measures the framerate of rendering, and the *visual accuracy benchmark*, which measures the image difference between GeoMipMapping and naive rendering. For both benchmarks, no fog and no overlay texture is rendered.

For the performance benchmark, two kinds of scenarios are performed:

- The first scenario is the flyover from the bottom-left corner to the top-right, whilst the camera is looking down a certain angle. The y-coordinate

and the front vector of the camera are fixed during this flyover.

- The second scenario is the 360° rotation while stationary.

For the visual accuracy benchmark, two kinds of scenarios are performed as well:

- The first scenario is a screenshot of a large section of the terrain at a great distance, once with LOD and once at full resolution.
- The second scenario is a screenshot of a smaller section of terrain but with a high camera zoom, once with LOD and once without.

5.2 Performance Benchmarks

The performance accuracy is computed by calculating the average FPS from the beginning of the flight or rotation to the end. The performance measurements of the naive renderer is not documented in this report, but was consistently around less than 1 FPS in both scenarios.

5.2.1 Flyover from Corner to Corner

Five flyovers were conducted from the bottom-left corner to the top-right corner with various velocities. Table 5.3 shows the FPS for each of the flyovers.

	Flyover 1	Flyover 2	Flyover 3	Flyover 4	Flyover 5	Average
FPS	59.83	60.14	60.35	60.08	57.89	59.65

Table 5.3: RMSE of the large terrain screenshots 1 to 5.

5.2.2 360° Rotation

Five rotations were conducted with various velocities. Table 5.4 shows the FPS for each of the rotations.

	Rotation 1	Rotation 2	Rotation 3	Rotation 4	Rotation 5	Average
FPS	60.61	60.57	61.62	60.08	61.11	60.65

Table 5.4: RMSE of the large terrain screenshots 1 to 5.

5.3 Visual Accuracy Benchmarks

For the visual accuracy benchmarks, the *root mean square error (RMSE)* of both images is computed :

$$RMSE = \sqrt{\frac{1}{mn}\sum_{x=1}^{m}\sum_{y=1}^{n}(A(x,y) - B(x,y))^2},$$

where m is the length of both images, n is the height of both images and A, B is the first and second image respectively [Mar23, p. 47]. The image difference and the RMSE are both computed using MATLAB.

Appendix B contains all screenshots for the visual accuracy benchmarks.

5.3.1 Large Terrain Screenshots

Five screenshots of the terrain were captured at various camera positions and angles, such that a large portion of the terrain was visible. The RMSE of every screenshot was computed, as shown in table 5.5.

	S. 1	S. 2	S. 3	S. 4	S. 5	Average
RMSE	3.94	3.1	2.59	1.96	2.32	2.78

Table 5.5: RMSE of the large terrain screenshots 1 to 5.

5.3.2 Low FOV Screenshots

Five screenshots of the terrain were captured at various camera positions and angles, such that only a small portion far away was visible due to the low FOV. The RMSE of every screenshot was computed, as shown in table 5.5.

	S. 1	S. 2	S. 3	S. 4	S. 5	Average
RMSE	4.82	5.71	4.78	5.3	4.49	5.02

Table 5.6: RMSE of the low FOV screenshots 1 to 5.

5.4 Memory Consumption

In most cases, the RAM and GPU memory consumption can be calculated manually for a given terrain size and block size.

5.4.1 RAM

The RAM consumption is mostly dependent on how many blocks (i.e. `GeoMipMappingBlock` instances) need to be managed. Generally, the larger the block size and the smaller the terrain size, the fewer blocks need to be managed.

5.4.2 GPU Memory

The main bottleneck of this implementation in terms of GPU memory is the heightmap texture, which takes up 2 bytes per height value. An alternative approach would be to support 1-byte grayscale heightmaps. However, this would limit the number of possible height values to 256 and therefore produce "blocky" looking terrain.

Memory consumption by the vertices and indices is quite low. The number of vertices that are loaded on the GPU is only $blockSize \times blockSize$.

5.4.3 Examples

Table 5.7 shows the GPU memory usage by the vertex buffers and index buffers for various block sizes. The size of the vertex buffer was calculated with $blockSize \times blockSize \times 2 \times 4$ and size of the index buffer was computed and printed directly in ATLOD.

Block size	Vertex buffer	Index buffer	Total
65	0.03 MB	0.12 MB	0.15 MB
129	0.13 MB	0.33 MB	0.46 MB
257	0.52 MB	1.02 MB	1.54 MB
513	2.1 MB	3.43 MB	5.53 MB

Table 5.7: Memory consumption by the vertex and index buffers for different block sizes.

Table 5.8 shows the GPU memory usage by the heightmap texture image for various heightmap sizes, calculated with $heightmapSize \times heightmapSize \times 2$.

Terrain size	Heightmap texture
2000×2000	8 MB
5000×5000	50 MB
16000×16000	512 MB

Table 5.8: Memory consumption by the heightmap texture on the GPU for various heightmap sizes.

Table 5.9 shows the memory usage by the blocks in GeoMipMapping's block list for various heightmap sizes and block sizes, calculated and printed in ATLOD direclty by multiplying the size of the list and `sizeof(GeoMipMappingBlock)`.

Block size	Heightmap size		
	2000×2000	5000×5000	16000×16000
65	0.08 MB	0.53 MB	5.45 MB
129	0.02 MB	0.13 MB	1.35 MB
257	4.31 KB	0.03 MB	0.33 MB
513	0.79 KB	7.12 KB	0.08 MB

Table 5.9: Memory consumption by the block list at different block sizes and heightmap sizes.

Chapter 6

Discussion

Overall, the implemented algorithm works decently well, despite lacking some features for it to be fully optimized. The configurability of the implementation allows for usage of the system for different applications and purposes.

In order to actually test the limits of the implemented algorithm, the performance measurements should be conducted again with stronger hardware. It is to be noted that the implementation still performs decently well, given the relatively weak hardware it was tested on.

A comparison with existing systems and game engines is difficult. ATLOD is developed specifically for terrain rendering, whilst game engines contain other components and often perform various tasks in the background, which hinders an accurate performance comparison.

Chapter 7

Conclusion

7.1 Potential Improvements

The GeoMipMapping implementation has some room for improvement:

- The view-frustum culling can be implemented more efficiently with a quadtree. The main problem with quadtree-based view-frustum culling is that in order to support non-square terrains, special care needs to be taken for the quadtree size. A simple solution would be to define the quadtree to have a side length of the next power of two larger than $\max\{terrainWidth, terrainHeight\}$ and to mark nodes as `null` in quadrants where there is no terrain.
- The performance can be further increased with *instanced rendering*. This would reduce the number of draw calls dramatically.
- The idea that a $(0, 0, 1, 0)$ border permutation is simply a $(1, 0, 0, 0)$ border permutation with a rotation of $-\pi/2$ can be applied to further reduce GPU memory usage. This could be achieved by allocating only the indices for the border permutations $(0, 0, 0, 0)$, $(1, 0, 0, 0)$, $(1, 1, 0, 0)$, $(1, 1, 1, 0)$ and $(1, 1, 1, 1)$ and then by simply rotating the flat mesh in the vertex shader, in addition to translating it.
- Another potential improvement is to extend the implemented algorithm with vertex morphing in order to reduce the popping artifacts.

7.2 Outlook for the Bachelor Thesis

There are several possible project ideas for the bachelor thesis which build upon this project and the topics behind it:

- Integration of a terrain LOD system in a game engine (e.g. Godot) or in a scene-graph library (e.g. SLProject).
- Development of a flight simulator, where the user can control the aircraft/-camera using gestures.

- Implementation of a streaming/paging-based terrain LOD algorithm, where multiple terrain instances are dynamically loaded and offloaded depending on the position. This would allow for (theoretically) infinite terrains.
- Implementation and benchmarking of additional terrain LOD algorithms. Some interesting and relevant algorithms that could be added are GPU-based Geometry Clipmaps, CDLOD and Concurrent Binary Trees.

Bibliography

[AH05] Arul Asirvatham and Hugues Hoppe. Terrain rendering using gpu-based geometry clipmaps. In *GPU Gems 2*. Addison-Wesley, 2005.

[And07] Johan Andersson. Terrain rendering in frostbite using procedural shader splatting. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, page 38–58, New York, NY, USA, 2007. Association for Computing Machinery.

[Bar] Sean Barrett. Stb libraries: Single-file public domain libraries for c/c++. `https://github.com/nothings/stb`.

[BN08] Eric Bruneton and Fabrice Neyret. Precomputed atmospheric scattering. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, EGSR '08, pages 1079–1086, Goslar, DEU, 2008. Eurographics Association.

[Cor] Omar Cornut. Dear imgui: Bloat-free immediate mode graphical user interface for c++ with minimal dependencies. `https://github.com/ocornut/imgui`.

[dB00] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping. In *The Web Conference*, 2000.

[DDKY21] Thomas Deliot, Jonathan Dupuy, Iijnen Kees, and Xiaoling Yao. Experimenting with concurrent binary trees for large scale terrain rendering. SIGGRAPH 2021 Advances in Real-time Rendering in Games course, 2021.

[Dup20] Jonathan Dupuy. Concurrent binary trees (with application to longest edge bisection). *Proc. ACM Comput. Graph. Interact. Tech.*, 3(2), aug 2020.

[dV20] Joey de Vries. *Learn OpenGL - Graphics Programming*. Kendall & Welling, 2020.

[DWS+97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes. In *Proceedings of the 8th Conference on Visualization '97*, VIS '97, pages 81–88, Washington, DC, USA, 1997. IEEE Computer Society Press.

[Fed] Federal Office of Topography swisstopo. swissALTI3D. `https://www.swisstopo.admin.ch/en/geodata/height/alti3d.html`.

[For] Sven Forstmann. Fast-quadric-mesh-simplification. `https://github.com/Zylann/godot_heightmap_plugin`.

[Gam] Epic Games. Landscape Overview — Unreal Engine 5.3 Documentation. `https://docs.unrealengine.com/5.3/en-US/landscape-overview/`.

[Gil] Marc Gilleron. Godot heightmap plugin. `https://github.com/Zylann/godot_heightmap_plugin`.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3), 2004.

[LKR+96] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-Time, Continuous Level of Detail Rendering of Height Fields. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 109–118, New York, NY, USA, 1996. Association for Computing Machinery.

[LWC+02] David Luebke, Benjamin Watson, Jonathan D. Cohen, Martin Reddy, and Amitabh Varshney. *Level of Detail for 3D Graphics*. Elsevier Science Inc., USA, 2002.

[Mar23] Marcus Hudritsch. Skript Grundlagen der Bildverarbeitung. Bern University of Applied Sciences, 2023.

[NAS13] NASA Shuttle Radar Topography Mission (SRTM). Shuttle radar topography mission (srtm) global. `https://doi.org/10.5069/G9445JDF`, 2013. Distributed by OpenTopography.

[Pet] Cory Petkovsek. Terrain3d. `https://github.com/TokisanGames/Terrain3D`.

[RHSS98] Stefan Röttger, Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Real-time generation of continuous levels of detail for height fields. *Journal of WSCG*, 6(1-3), 1998.

[Sal15] Jose Salvatierra. 4.2. blending, aliasing, and fog. `https://opengl-notes.readthedocs.io/en/latest/topics/texturing/aliasing.html`, 2015.

[Sav17] Mike J Savage. Geometry clipmaps: simple terrain rendering with level of detail. `https://mikejsavage.co.uk/blog/geometry-clipmaps.html`, 2017.

[Seo] JangKyu Seo. Unity hlod system. `https://github.com/Unity-Technologies/HLODSystem/`.

[Str09] Filip Strugar. Continuous distance-dependent level of detail for rendering heightmaps. *J. Graphics, GPU, & Game Tools*, 14:57–74, 01 2009.

[Tec] Unity Technologies. Unitymeshsimplifier.

[Ulr02] Thatcher Ulrich. Rendering Massive Terrains using Chunked Level of Detail Control. `http://tulrich.com/geekstuff/chunklod.html`, 2002.

[Wid12] Mattias Widmark. Terrain in battlefield 3: A modern, complete and scalable system. `https://www.slideshare.net/DICEStudio/terrain-in-battlefield-3-a-modern-complete-and-scalable-system`, 2012.

Appendix A

DEM Preprocessing

The following steps can be performed to convert a GeoTIFF file or Esri ASCII grid file into a 16-bit grayscale PNG heightmap image using the GIS software tool QGIS:

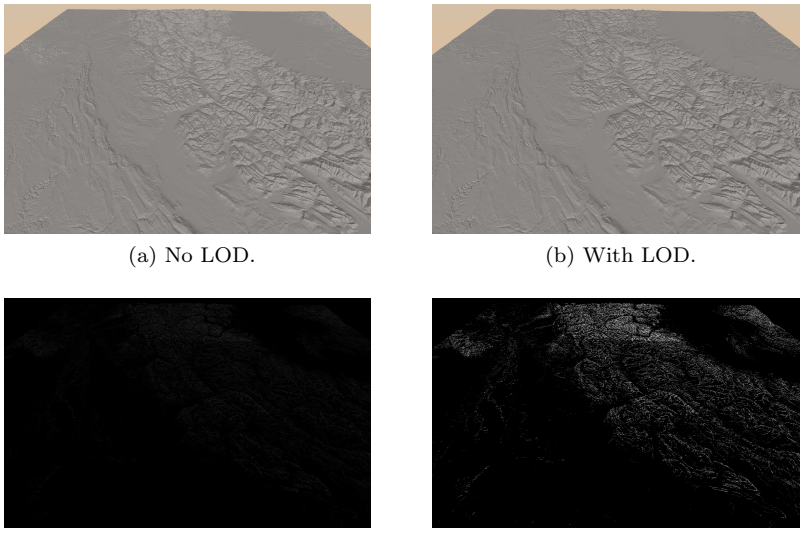
1. Open the GeoTIFF or Esri ASCII grid file with QGIS
2. Select “Raster” → “Conversion” → “Translate (Convert Format)...”
3. Select your heightmap as the input layer
4. “Output data type”: UInt16
5. “Converted”: the path for the new heightmap with “.png” postfixed.
6. Press “Convert”

Appendix B

Visual Accuracy Benchmarking Images

B.0.1 Large Terrain Screenshots

Large Terrain Screenshot 1



(a) No LOD.

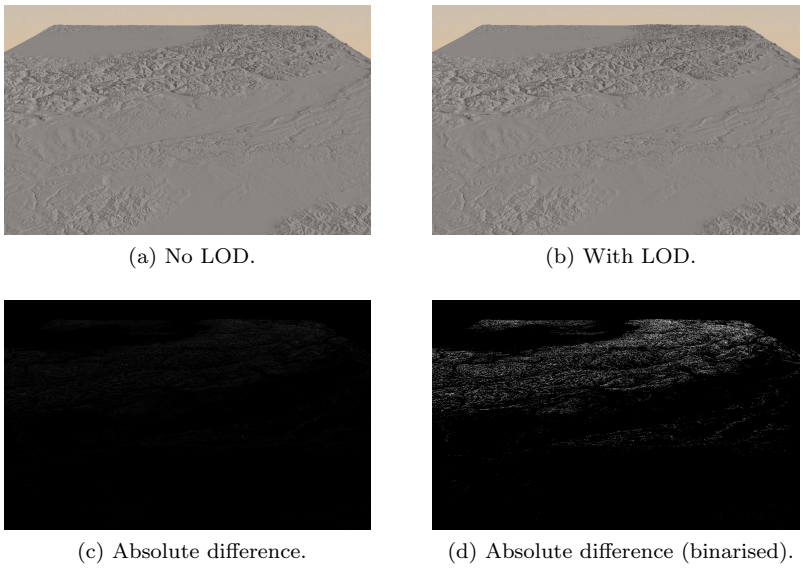
(b) With LOD.

(c) Absolute difference.

(d) Absolute difference (binarised).

Figure B.1: Screenshot showcasing the screenshot of a large section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b), and the binarised absolute difference (d) of (c). The computed RMSE is 3.94.

Large Terrain Screenshot 2



(a) No LOD.

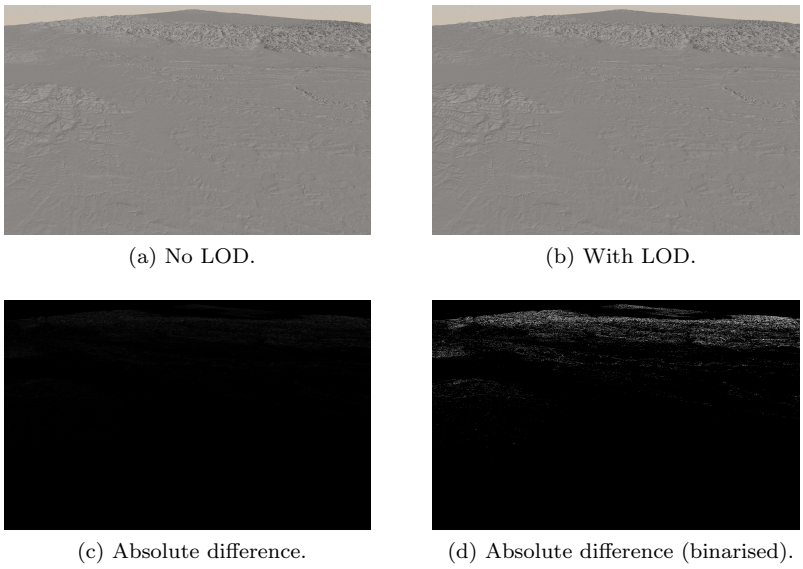
(b) With LOD.

(c) Absolute difference.

(d) Absolute difference (binarised).

Figure B.2: Screenshot showcasing the screenshot of a large section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b), and the binarised absolute difference (d) of (c). The computed RMSE is 3.1.

Large Terrain Screenshot 3



(a) No LOD.

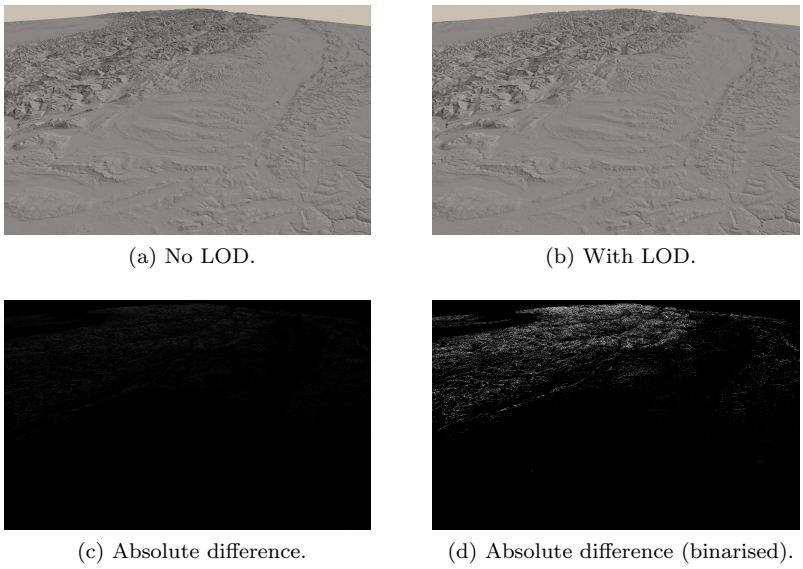
(b) With LOD.

(c) Absolute difference.

(d) Absolute difference (binarised).

Figure B.3: Screenshot showcasing the screenshot of a large section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b), and the binarised absolute difference (d) of (c). The computed RMSE is 2.59.

Large Terrain Screenshot 4



(a) No LOD.

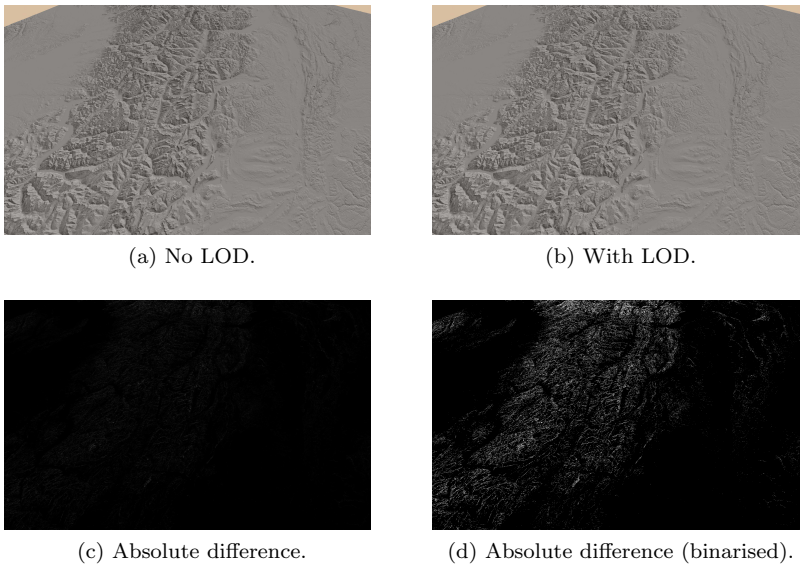
(b) With LOD.

(c) Absolute difference.

(d) Absolute difference (binarised).

Figure B.4: Screenshot showcasing the screenshot of a large section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b), and the binarised absolute difference (d) of (c). The computed RMSE is 1.96.

Large Terrain Screenshot 5



(a) No LOD.

(b) With LOD.

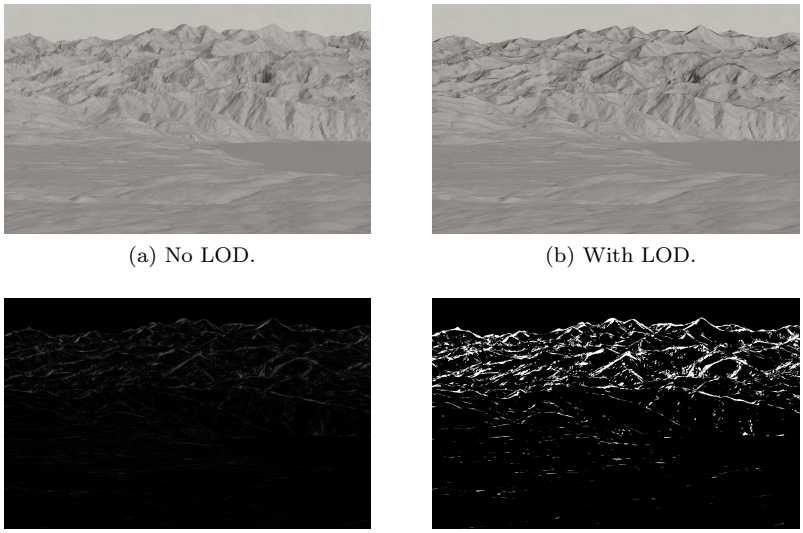
(c) Absolute difference.

(d) Absolute difference (binarised).

Figure B.5: Screenshot showcasing the screenshot of a large section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b), and the binarised absolute difference (d) of (c). The computed RMSE is 2.32.

B.0.2 Low FOV Screenshots

Low FOV Screenshot 1



(a) No LOD.

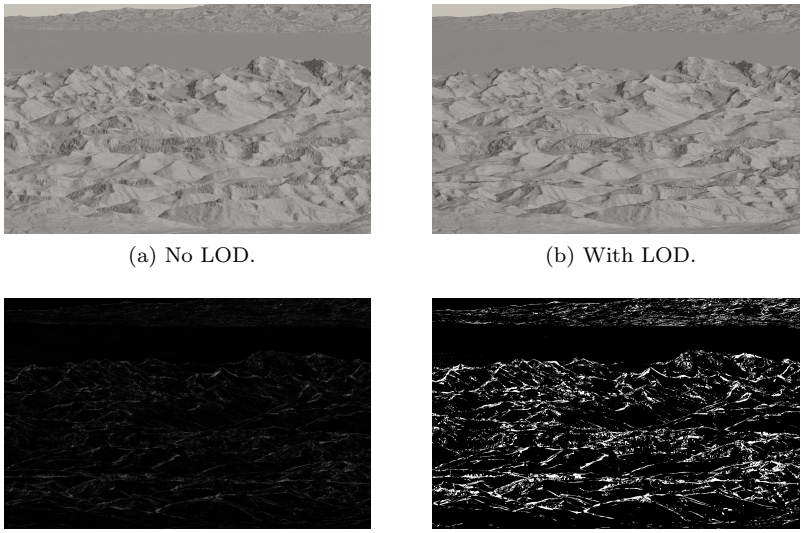
(b) With LOD.

(c) Absolute difference.

(d) Absolute difference (binarised).

Figure B.6: Screenshot showcasing the screenshot of a small section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b) and the binarised absolute difference (d) of (c). The FOV is set to 6° and the computed RSME is 4.82.

Low FOV Screenshot 2



(a) No LOD.

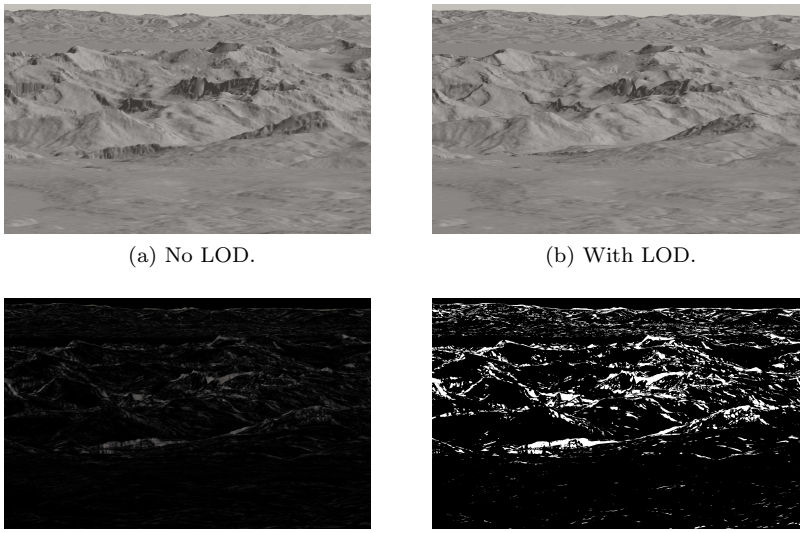
(b) With LOD.

(c) Absolute difference.

(d) Absolute difference (binarised).

Figure B.7: Screenshot showcasing the screenshot of a small section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b) and the binarised absolute difference (d) of (c). The FOV is set to 3° and the computed RSME is 5.71.

Low FOV Screenshot 3

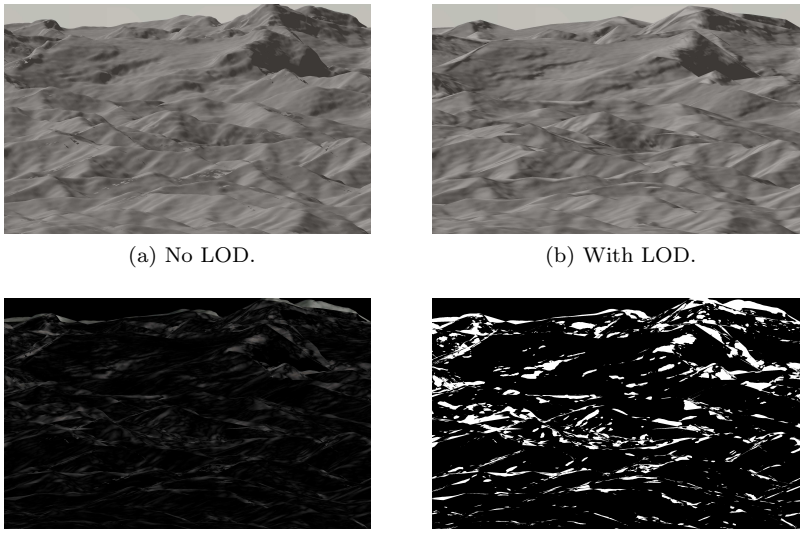


(a) No LOD. (b) With LOD.

(c) Absolute difference. (d) Absolute difference (binarised).

Figure B.8: Screenshot showcasing the screenshot of a small section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b) and the binarised absolute difference (d) of (c). The FOV is set to 2° and the computed RSME is 4.78.

Low FOV Screenshot 4



(a) No LOD.

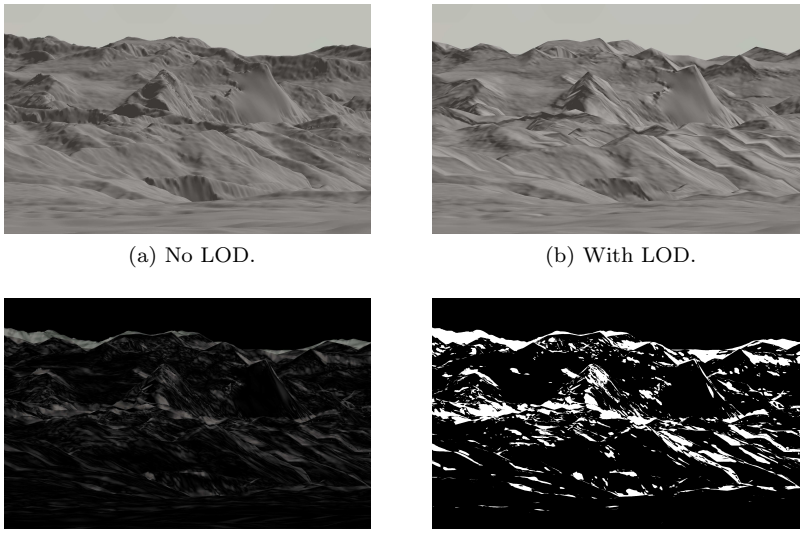
(b) With LOD.

(c) Absolute difference.

(d) Absolute difference (binarised).

Figure B.9: Screenshot showcasing the screenshot of a small section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b) and the binarised absolute difference (d) of (c). The FOV is set to 1° and the computed RSME is 5.3.

Low FOV Screenshot 5



(a) No LOD.

(b) With LOD.

(c) Absolute difference.

(d) Absolute difference (binarised).

Figure B.10: Screenshot showcasing the screenshot of a small section of the terrain with no LOD (a), with LOD (b), the absolute difference (c) between (a) and (b) and the binarised absolute difference (d) of (c). The FOV is set to 1° and the computed RSME is 4.49.

Erklärung der Studierenden
Déclaration des étudiant-e-s

Selbständige Arbeit / *Travail autonome*

Ich bestätige mit meiner Unterschrift, dass ich meine vorliegende Projektarbeit selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.) und anderen Hilfsmittel, die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt. Sämtliche Inhalte, die nicht von mir stammen, sind mit dem genauen Hinweis auf ihre Quelle gekennzeichnet.

Par ma signature, je confirme avoir effectué mon présent travail de projet de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.) et autres ressources qui m'ont fortement aidé-e dans mon travail sont intégralement mentionnées dans l'annexe de ma thèse. Tous les contenus non rédigés par mes soins sont dûment référencés avec indication précise de leur provenance.

Name/*Nom*, Vorname/*Prénom* ..... Tabakovic, Amar

Datum/*Date* ..... 19. Januar 2024

Unterschrift/*Signature* .....

Dieses Formular ist dem Bericht beizulegen.
Ce formulaire doit être joint au rapport.