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

December 18, 2023

Abstract

Rendering terrains is a central task for video games, geographic information systems and simulators, but also computationally expensive. Optimizations, one of which is the level of detail (LOD), are necessary in order to ensure appropriate performances. This project aims to give an overview of the topic of terrain rendering. As part of this project, a demo terrain renderer (ATLOD) was implemented in C++ and OpenGL. ATLOD currently supports naive brute-force rendering, GeoMipMapping, TODO.

Contents

Chapter 1

Introduction

In the field of 3D computer graphics, rendering is one of the central tasks. Many practical applications of 3D computer graphics make use of terrains, such as flight simulators, open-world video games, and Geographic Information Systems (GIS) [1, p. 185]. At the same time, rendering large and constantly visible objects, such as the terrain, is computationally expensive and optimizations are necessary in order to avoid performance deficiencies.

One area which offers potential for optimizations is the *level of detail (LOD)* of objects. The concept of LOD is based on the idea that the farther away an object is, the fewer details are going to be visible to the human eye.

The problem of rendering terrains spawned numerous algorithms and approaches specifically for this purpose.

1.1 Goals of this Project

The main goal of this project is to gain an overview over the topic of terrain rendering, in theory as well as in practice. This includes introducing the basics of terrain programming, analyzing and comparing existing approaches and algorithms, and researching what approaches are used in the real world. For this purpose, a demo application (named *ATLOD*) that utilizes some of the analyzed algorithms is developed in C++ and OpenGL.

1.2 Structure of the Report

This report is structured as follows:

- Chapter 2 gives an overview of work that has already been conducted in the area of terrain LOD and relevant literature for this project.
- Chapter 3 describes the basics of terrain rendering, such as terrain data representations and potential problems during rendering.
- Chapter 4 introduces the reader to a selection of terrain LOD algorithms. Each algorithm is described in a high-level manner.

- Chapter 5 Lists a few real-world examples where terrain LOD is being used, such as game engines or geographic information systems. The algorithms are presented in a high-level manner and in chronological order of their publication date.
- Chapter 6 describes ATLOD, the demo application for terrain rendering implemented as part of this project. An overview of the functionalities is given and the main approaches and design decisions are discussed. The concrete implementations of the algorithms are described in detail.
- Chapter 7 TODO

Chapter 2

Existing Work and Literature

2.1 Research Articles and Publications

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 more detail in chapter "Algorithms and Approaches for Terrain LOD" are highlighted in **bold**:

- "Real-Time, Continuous Level of Detail Rendering of Height Fields" [2] by Lindstrom *et al.* in 1996.
- **"ROAMing Terrain: Real-time Optimally Adapting Meshes"** [3] by Duchaineau *et al.* in 1997.
- "Real-Time Generation of Continuous Levels of Detail for Height Fields" [4] by Röttger *et al.* in 1998.
- **"Fast Terrain Rendering Using Geometrical MipMapping"** [5] by de Boer in 2000.
- "Rendering Massive Terrains using Chunked Level of Detail Control" [6] by Ulrich in 2002.
- "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids" [7] by Hoppe and Losasso in 2004 and the follow-up GPU-based "Terrain Rendering Using GPU-Based Geometry Clipmaps" by Asirvatham and Hoppe TODO Cite in 2005.
- "Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD)" [8] by Strugar in 2009.
- "Concurrent Binary Trees (with application to longest edge bisection)" TODO cite by Dupuy in 2020.

2.2 Level of Detail for Computer Graphics

Level of Detail for Computer Graphics by Luebke *et al.* [1] is a reference book for the topic of LOD published in 2002. The book builds on top of years of research in the area of LOD and provides an overview to many LOD techniques. For this project, chapter 7 "Terrain Level of Detail" of the book is especially relevant, as it dives into the topic of LOD for terrains specifically. It covers basic approaches and techniques for terrain LOD, common problems that can arise during rendering of terrains and some solutions to them, and a catalog of terrain LOD algorithms.

2.3 Focus on 3D Terrain Programing

Focus on 3D Terrain Programming by Trent Polack [9] is a book on terrain programming published in 2002. Part one of the book introduces the reader to the basics of terrains, such as height maps and texturing. Part two then covers some more advanced topics, including a selection of terrain LOD algorithms. The presented LOD algorithms are ROAM [3], Röttger's quadtree-based algorithm [4], and GeoMipMapping [5]. The book also includes demo source code in C++ and OpenGL.

Despite having been released more than 20 years ago, some of the basic ideas of terrain rendering are still valid even today.

2.4 Virtual Terrain Project

The *Virtual Terrain Project* [10] was a project run from 2001 to 2013 that consisted of a collection of software, information and resources on terrain modelling and rendering, including a large overview of publications and implementations related to terrain LOD algorithms.

Chapter 3

Basics of Terrain Rendering

3.1 Terrain Data Representation

3.1.1 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[1]. Positions are always spaced evenly in a grid-like manner, but the distance between any two neighboring positions 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. For a grayscale image, up to 256 height values can be used and for an RGB image, more than 16 million height values are supported. Looking up a height value for a given (x, z)-position is easy as well, which consists of a simple lookup at the given position in the image. Figure 3.1 shows a 2000×2000 heightmap of the mountain Dom in Valais, Switzerland.

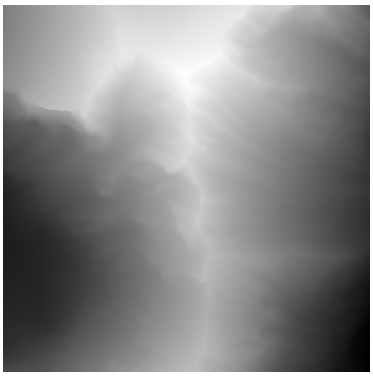


Figure 3.1: 2000×2000 heightmap of the mountain Dom in Valais, Switzerland retrieved from SwissTopo [11].

[1] We always denote y for the up direction except if explicitly stated otherwise.

3.1.2 Triangulated Irregular Networks

An alternative to the heightmap is the *triangulated irregular network (TIN)* data structure. A TIN consists of a collection of 3D vertices, where the arrangement of vertices can be irregular. Figure 3.2 shows an example of a TIN.

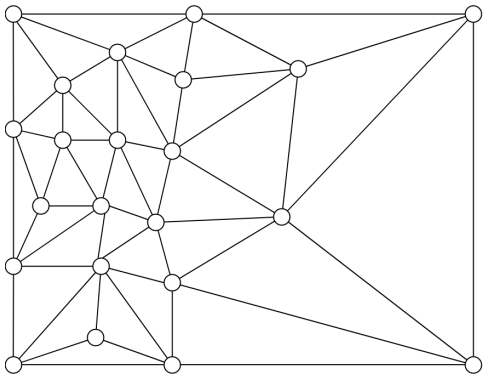
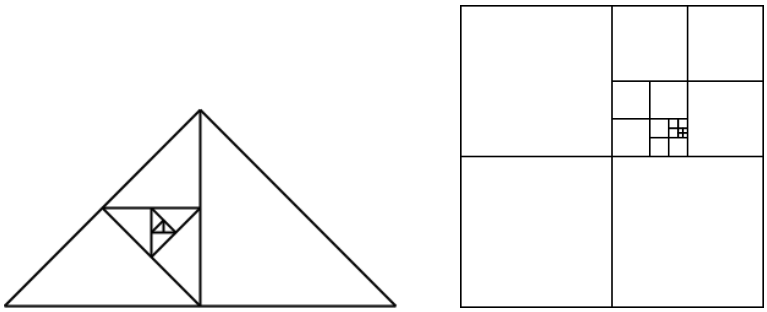


Figure 3.2: Example of a TIN. Note that the left area represents a terrain area with many changes (e.g. mountains, hills, etc.), and the right area represents an area with few changes (e.g. flat areas).

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, 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.

3.2 Bintrees and Quadtrees

Binary triangle trees (bintrees) and *quadtrees* are recursive data structures based on triangles and quads respectively. A bintree consists of up to two child triangles, both of which in turn also consist of up to two child triangles each, and so on. 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 on. Figure 3.3 shows an example of a bintree and a quadtree.



(a) (b)

Figure 3.3: Example of a bintree (a) and a quadtree (b).

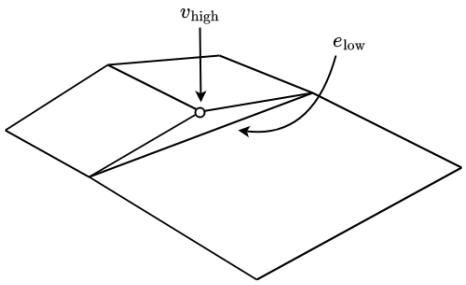
The main advantage of bintrees and quadtrees is that LOD can be modelled very naturally with them. Bintree/quadtree sections with few children correspond to a low LOD and vice versa for bintree/quadtree sections with many children.

3.3 Potential Problems During Terrain Rendering

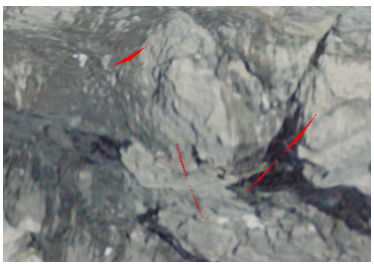
While terrain LOD algorithms dramatically improve the performance of terrain rendering, there are certain faults that can occur during rendering.

3.3.1 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 3.4.



(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 3.4: 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, cuasing the higher and lower LOD meshes to be connected seamlessl (in figure 3.4 vertex v_{high}).
- Inserting an extra vertex at the border edge of the lower LOD mesh [1, p. 194] (in figure 3.4 on top of vertex v_{high}). The disadvantage of this is that an extra vertex needs to get created.
- By force splitting the terrain mesh to get a more continuous mesh [1, p. 193].

3.3.2 Popping

The phenomenon of *popping* occurs when the camera is moving and the transition of the terrain's 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 by introducing *vertex morphing* [5, 7, 8], i.e. by animating the transition of one LOD level to the next seamlessly through interpolation.

Chapter 4

Algorithms and Approaches for Terrain LOD

4.1 ROAM

ROAM (short for **R**eal-time **O**ptimally **A**dapting **M**eshes) is a terrain LOD algorithm developed by Duchaineau *et al.* [3] published in 1997. ROAM represents the terrain mesh using bintrees and performs triangle splits and merges for generating and removing detail. The splits and merges happen mainly on the CPU, which makes ROAM a mainly CPU-based approach.

4.1.1 General Idea

The central idea of the algorithm is to use temporal coherence: the mesh from a previous frame $\mathbf{T}_{f-1}$ is used to compute the mesh of the current frame $\mathbf{T}$, rather than building up the mesh from ground up for each frame. 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). The elements of the priority queues are ordered by various geometric error metrics, which are explained in the subsection "Error Metrics" of this report. 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. ROAM is designed as a greedy algorithm, meaning it will always performs the most optimal splits/merges for each frame.

4.1.2 Error Metrics

The original ROAM paper mentions various error metrics for the priority queue ordering, which are explained in the following paragraphs.

Wedgies *Wedgies* are nested bounding volumes around triangles that are computed while building the initial mesh at the beginning of the algorithm. A wedgie

is defined to contain the entire x and z extent[1] of a triangle and the height y including some additional space above and below the highest and lowest points, respectively.

Geometric Screen Distortion Another metric is the distance between where a node is supposed to be on the screen and where the algorithm placed the node. The maximum of all distances is calculated and used as the base priority metric of the algorithm.

4.1.3 Other Optimizations

View-frustum Culling An optimization that is mentioned is view-frustum culling. In each frame, various flags are updated during the recursive traversal of the bintree. These flags indicate whether a wedgie is inside the view-frustum fully, partially, or not at all.

Incremental Triangle Stripping The ROAM paper reports using *incremental triangle stripping* for optimizing the performance of the rendering. This is simply refers to using triangle strips for rendering. which is supported by all major graphics APIs.

4.1.4 Conclusion

The runtime of ROAM is independent of the screen resolution and is proportional to the number of triangle changes per frame.

4.2 GeoMipMapping

Geometrical Mipmapping (GeoMipMapping) is a terrain LOD approach developed by de Boer [5] in the year 2000. It applies the idea of texture mipmapping to terrain rendering. In this section, all presented ideas are from the original paper, unless noted otherwise.

4.2.1 General Idea

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.

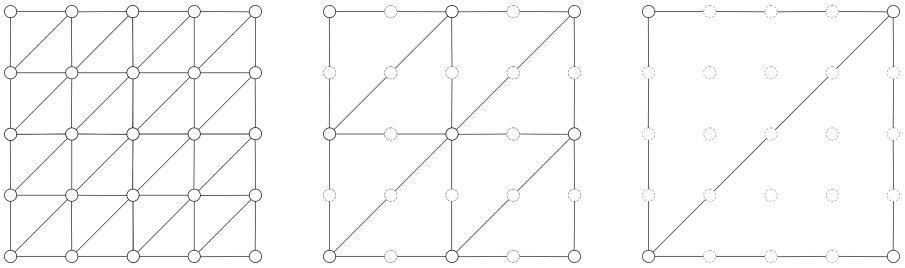
This is achieved by splitting up the terrain into so-called *blocks* (also called *patches*) of a fixed width $2^n + 1$ for an arbitrary $n \in \mathbb{N}$. Each block has a LOD level[2] $0 \leq l \leq n$ that changes dynamically at runtime. At load time, every block is computed at every LOD level and subsequently stored, e.g. on an index buffer. Each such representation of a block at a specific LOD level is called a

[1] The original ROAM paper uses z for the up direction. For the sake of consistency with the rest of this report, we will use y as the up direction here.

[2] The original GeoMipMapping paper uses 0 to denote the maximum LOD level and vice-versa for the minimum LOD level. In order to avoid any confusion with the term *LOD*, this report denotes 0 as the minimum LOD level and vice versa for the maximum LOD level.

GeoMipMap. For each GeoMipMap, the number of vertices on one side is $2^l + 1$ and the number of quads is 2^{2l}.

For example, the GeoMipMaps of a 5×5 terrain block contain $2^2 + 1 = 5$ vertices on one side at the maximum LOD level 2, $2^1 + 1 = 3$ vertices at LOD level 1 and $2^0 + 1 = 2$ vertices at the minimum LOD level 0. As for the number of quads, at LOD level 2 there are $2^4 = 16$ quads, at LOD level 1 there are $2^2 = 4$ quads and at LOD level 0 there is $2^0 = 1$ quad. Figure 4.1 illustrates the above example.



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

Figure 4.1: Example of each GeoMipMap of a 5×5 block. The omitted vertices of lower LOD GeoMipMaps are marked as dotted circles.

4.2.2 View-frustum Culling

The organisation of the terrain into blocks allows for easy view-frustum culling, which dramatically decreases the number of vertices that need to get rendered.

4.2.3 LOD Selection

The LOD for each block is selected at runtime and is based on a

4.2.4 Avoiding Cracks

GeoMipMapping avoids cracks by checking whether the current block has a higher LOD than the bordering block and if this is the case, it omits the vertices that would cause the crack. The vertex omission is performed by rendering the bordering row/column of the current block as triangle fans, as shown in figure 4.2.

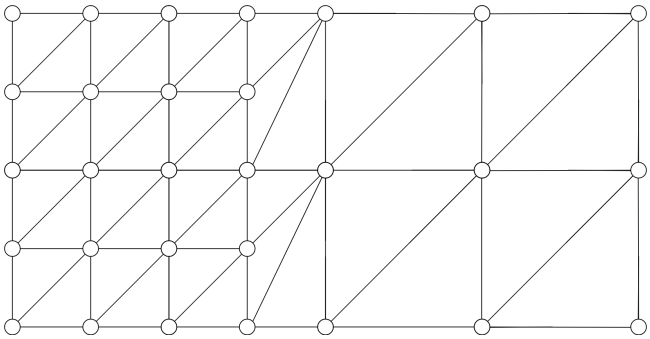
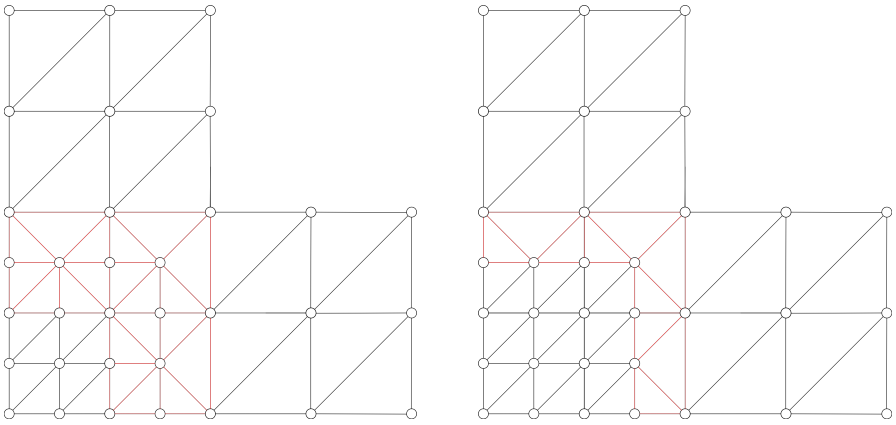


Figure 4.2: Example of GeoMipMapping's crack avoidance between a LOD 2 and a LOD 1 GeoMipMap of two 5×5 blocks as described in the original paper.



(a) Using the entire 3×3 border subblocks for the triangle fans.

(b) Using only the top-most 2×3 subblocks and the right-most 3×2 subblocks for the triangle fans.

Figure 4.3: Two alternative triangle fan-based methods to avoid cracks between a LOD 2 and two LOD 1 GeoMipMaps of three 5×5 blocks. The border subblocks are marked in red.

Regardless of the rendering approach, a neighborhood structure consisting of the LOD levels of the left, right, top and bottom neighboring blocks needs to be stored per block, so that the algorithm knows how to perform the vertex omission. The LOD levels of each block change continually each frame, which means that the neighborhood structure of each block also gets updated each frame.

4.2.5 Other Optimizations

Vertex Morphing GeoMipMapping can be extended with vertex morphing in order to decrease popping.

4.3 Geometry Clipmaps

TODO

4.4 GPU Tessellation Shaders

Chapter 5

Terrain LOD in Real-world Systems

5.1 Game Engines

5.1.1 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 [12] written in C++ for Godot 4. The LOD approach used in this extension is based on geometry clipmaps by Hoppe and Losasso [7]. The concrete implementation of the geometry clipmap mesh code was created by Mike J Savage [13].

Another extension for terrains is Godot Heightmap Plugin by Marc Gilleron [14] written in GDScript and C++. The extension uses a quadtree-based approach for terrain LOD.

5.1.2 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 terrain LOD is used for the built-in terrain in Unity.

Nonetheless, there exists an open-source library for hierarchical LOD in Unity called HLODSystem developed by JangKyu Seo at Unity TODO citation . 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, an open-source mesh simplifier de-

veloped by TODO that utilizes the fast quadratic mesh simplification algorithm developed by TODO citation.

In addition to the mentioned open-source libraries, a terrain LOD approach was published by Jonathan Dupuy at Unity in 2020.

5.1.3 Unreal Engine

Unreal Engine is another cross-platform game engine written in C++ and features an integrated terrain system called the Landscape system. The technical documentation of Unreal Engine 5 mentions utilising GeoMipMapping for handling LOD for landscapes [15]. GeoMipMapping is a terrain LOD approach developed in 2000 by de Boer [5].

5.1.4 Frostbite

Frostbite is a closed-source game engine developed by DICE and is known for the *Battlefield* series. During the Game Developers Conference 2012, DICE presented the terrain system of *Battlefield 3*, which was developed with their Frostbite 2 engine. They mention a quadtree-based terrain LOD system and describe several optimizations regarding paging and streaming of terrain. TODO cite.

5.2 Geographic Information Systems

TODO: Check out Google maps, Google Earth, other GIS TODO: Maybe look at flight simulators (if information available)

Chapter 6

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.

6.1 Preliminaries

6.1.1 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.
- STB: STB is a collection of header-only libraries developed by Sean Barrett TODO cite. 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 TODO.

6.1.2 Chosen Algorithms

The chosen algorithms and their reasons for implementing them are the following:

Naive Brute-force Algorithm The naive brute-force algorithm consists of simply reading in all height values from the heightmap as vertices and rendering them directly to the screen. The main reason for implementing the naive brute-force algorithm is to motivate the usage of terrain LOD algorithms by showing the difference in performance compared to the optimized LOD approaches.

GeoMipMapping The main reason for implementing GeoMipMapping is its simplicity.

6.2 Height and Texture Data

6.2.1 Data Sources

SwissTopo

SRTM

6.2.2 Supported Formats

The following file formats are supported for loading heightmaps:

- `.png` and `.jpg`: Heightmaps can be loaded as PNG and JPG images. This is implemented using `stb_image.h`.
- `.asc`: Heightmaps as ASCII grids are supported. Digital elevation data from the Shuttle Radar Topography Mission (SRTM) is delivered in ASCII grid format.
- `.xyz`: the XYZ format is based on SwissTopo

6.3 Basic Setup and Architecture

6.3.1 Overview

TODO High-level class diagram

6.3.2 Shaders

6.3.3 Camera

6.3.4 Heightmaps

6.3.5 Base Terrain

Each terrain LOD algorithm is encapsulated in a class, which inherits from the base terrain class `Terrain`. The base terrain is structured as shown in listing TODO.

6.4 Naive Brute-force Algorithm

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

The method `loadBuffers()` loads the height data from the heightmap directly into a vertex buffer with the ID `terrainVBO` and the indices into an index (i.e. element) buffer named `terrainEBO`. 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 6.1 shows the organization of indices for rendering the terrain as triangle strips.

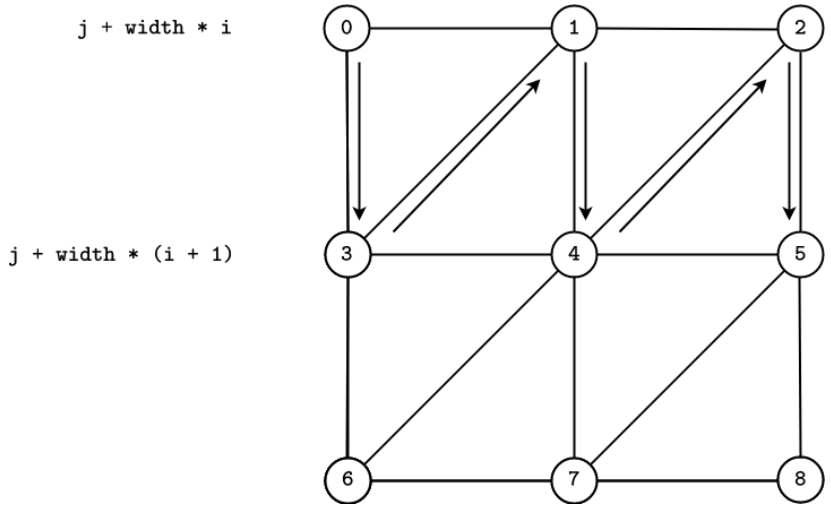


Figure 6.1: Example of a terrain layout for triangle strips. The looping index `i` goes from 0 to the terrain height and `j` from 0 to the terrain width. The final indices to be rendered are `0,3,1,4,2,5,RESTART,3,6,4,7,5,8,RESTART`.

6.5 GeoMipMapping

ATLOD's GeoMipMapping implementation contains the basic GeoMipMapping features described in sections 1 and 2 in the original paper. The algorithm is encapsulated in the class `GeoMipMapping` and differs in certain aspects:

- This GeoMipMapping implementation makes use of vertex and index buffers, which are more efficient than rendering vertices in immediate mode. The organisation of vertices and indices is described in more detail in the subsection "Vertex and Index Organisation".
- The vertex omission for avoiding cracks is implemented somewhat differently from the original paper. Instead of drawing triangle fans as shown in figure 4.2, this implementation uses the alternative approach shown in figure 4.3.

6.5.1 Data Structures

6.5.2 Vertex and Index Organisation

Vertices are loaded into the vertex buffer in the method `loadBuffers()`.

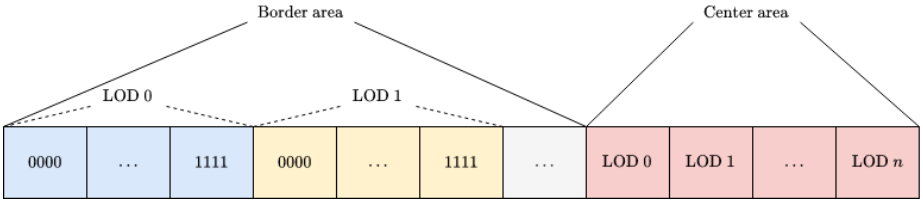
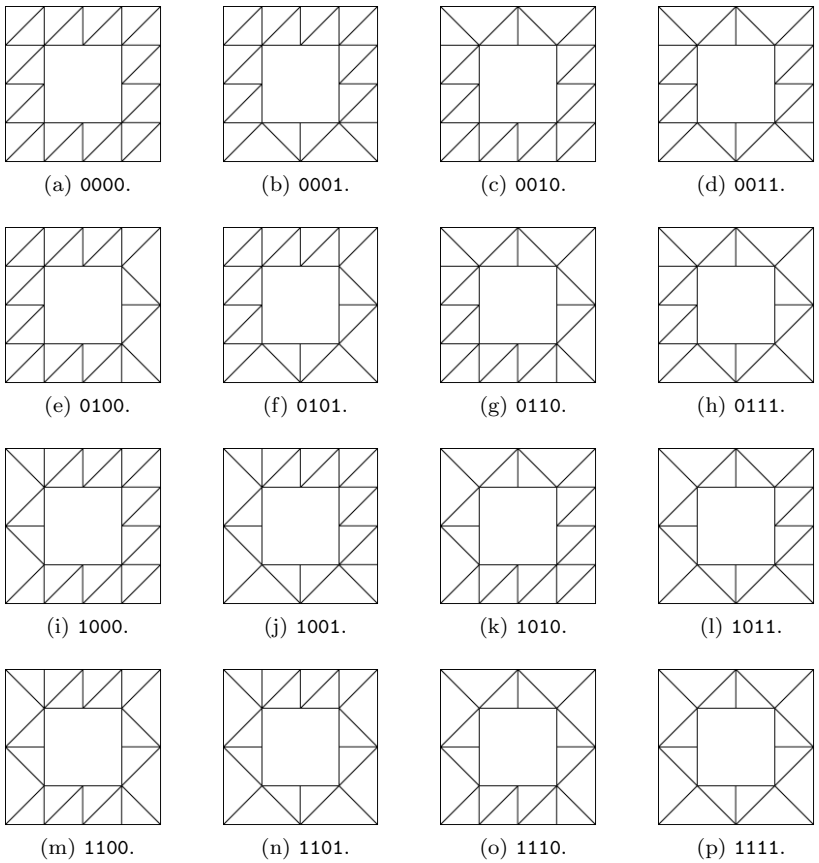


Figure 6.2: The index buffer organisation of a single block for the ATLOD's GeoMipMapping implementation. The variable n is from the block size equation $2^n + 1$ and corresponds to the maximum LOD level.

6.5.3 Avoiding Cracks



(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 6.3: Every possible border subblock configuration for a LOD 2 GeoMipMap of a 5×5 block. The bits (read from left to right) represent the left, right, top and bottom borders and are set to 1 if the bordering block on the corresponding side has a lower LOD, and 0 otherwise. The center subblocks have been omitted from the illustration.

6.5.4 Rendering

The method `render()` that is called each frame updates and renders the GeoMipMapping terrain. It consists of two sequential for-loop passes over all blocks of the terrain.

6.5.5 Conclusion

The GeoMipMapping implementation supports

The implementation still suffers from some weaknesses:

- The mesh preparation stage, where the indices of each block at every LOD level and border configuration are generated, takes a significant amount of

time, especially for larger terrains. An alternative approach would be to dynamically allocate and deallocate vertex and index buffers at runtime, e.g. when the LOD of a block changes. This, however, would have an impact on the runtime performance.

- On low LOD levels, a block consists of a low number of vertices. This means that the lighting calculated by the Phong shading relies on a low number of normals, which causes the lighting to look coarse at great distances. This phenomenon can be hidden by rendering with a distance fog, or by choosing a smaller block size. A potential solution for this would be to calculate the Phong shading not with per-vertex normals, but with a normal map.
-

6.6 Geometry Clipmaps

6.7 OpenGL Tessellation Shaders

Chapter 7

Results

7.1 Experimental Setup

For testing out the algorithms, I used a MacBook etc. TODO

Chapter 8

Discussion and Conclusion

8.1 Further Work

8.2 Reflexion

Bibliography

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

[2] 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. doi:10.1145/237170.237217.

[3] 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.

[4] 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. URL: http://hdl.handle.net/11025/15845.

[5] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping. In *The Web Conference*, 2000. URL: https://api.semanticscholar.org/CorpusID:7296927.

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

[7] Frank Losasso and Hugues Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3), 2004. doi:10.1145/1015706.1015799.

[8] Filip Strugar. Continuous distance-dependent level of detail for rendering heightmaps. *J. Graphics, GPU, & Game Tools*, 14:57–74, 01 2009. doi:10.1080/2151237X.2009.10129287.

[9] Trent Polack and Willem H. de Boer. *Focus on 3D Terrain Programming*. Premier Press, 2002.

[10] Virtual terrain project. https://vterrain.org.

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

[12] Cory Petkovsek. Terrain3d. `https://github.com/TokisanGames/Terrain3D`, 2023.

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

[14] Marc Gilleron. Godot heightmap plugin. `https://github.com/Zylann/godot_heightmap_plugin`, 2023.

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