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 17, 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. A demo terrain renderer (named ATLOD) was developed. The implemented algorithm is based on GeoMipMapping [1] and GPU-based Geometry Clipmaps [2].

Contents

List of Tables

List of Figures

Listings

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) [4, 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 survey and evaluation 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 composed. Afterwards, a demo terrain renderer is developed, using the ideas from one or more of the evaluated algorithms.

This project is restricted to rendering terrains from heightmaps without streaming/paging.

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 $\mathbb{R}^3$.

If at any point the report contains mathematical notation which is not described here, mathematical notation as commonly found in computer graphics is used.

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 resolu'ion 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 given.
- Chapter 4 describes the demo terrain renderer (named ATLOD) which was developed. The basic features and the implementation details of the algorithms are presented.
- 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.
- 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.
- Chapter 7 TODO

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.

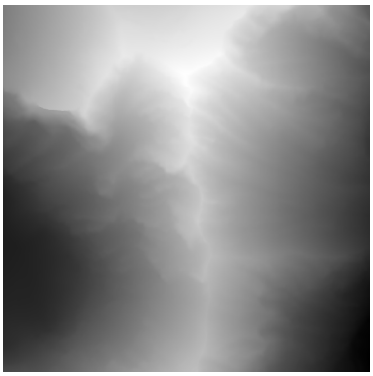


Figure 2.1: 2000×2000 heightmap of the mountain Dom in Valais, Switzerland retrieved from SwissTopo [3].

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.

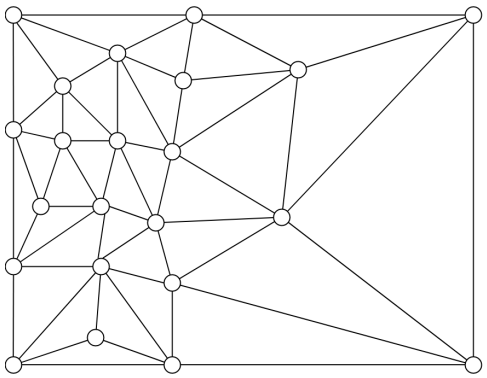


Figure 2.2: Example of a TIN. Note that the left area represents a terrain area with many changes (e.g. mountains, hills, etc.), and the right area represents an area with few changes (e.g. flat areas).

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 [4]. 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 with heightmaps, such as [5, 1, 6, 7, 2, 8, 9, 10], 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. Bintrees and quadtrees are mostly found in historical algorithms, such as [11] and [5], but have recently been revitalized in [10].

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.

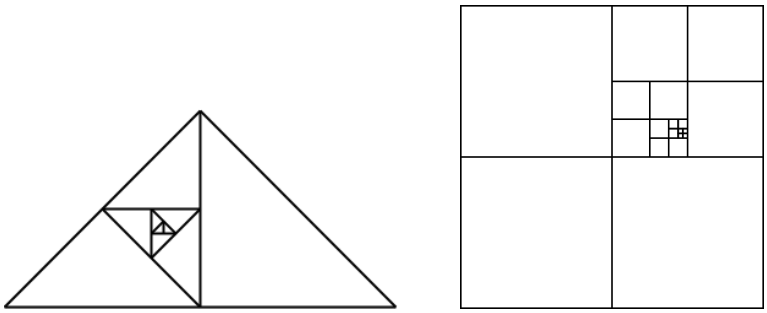


Figure 2.3: Example of a bintree (a) and a quadtree (b).

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.

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 [11, 5, 1, 7, 9]. 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.

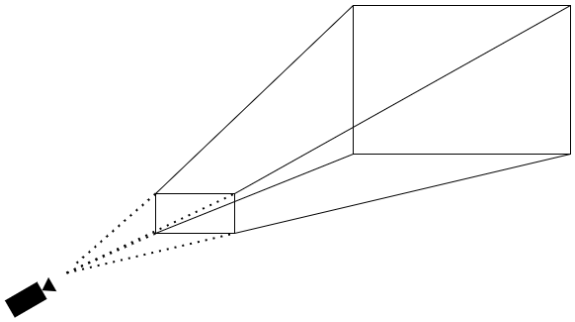


Figure 2.4: Example of a view-frustum

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 [1, 7, 9] 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.

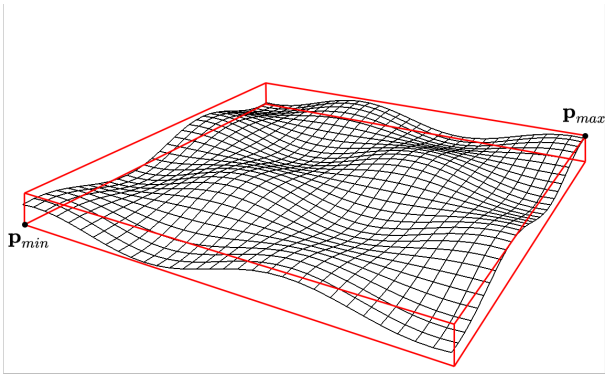


Figure 2.5: Example of a terrain block with its AABB defined by $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$, marked in red.

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 that contains all bounding volumes 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 more memory consumption. Figure 2.6 shows an example of quadtree-based view-frustum culling.

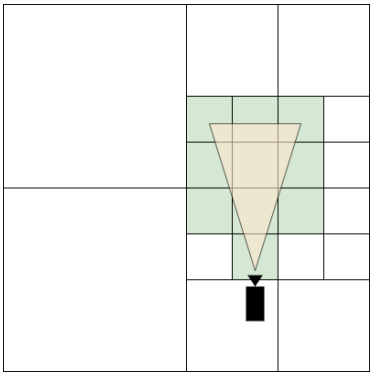


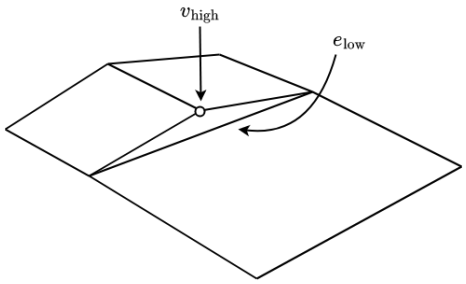
Figure 2.6: Example of view-frustum culling with a quadtree viewed from the top. The view-frustum is marked in yellow and blocks that intersect the view-frustum are marked in green.

2.4 Potential Problems During Terrain Rendering

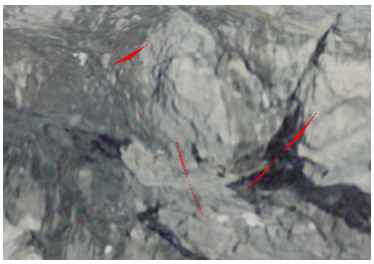
While terrain LOD algorithms dramatically improve the performance of terrain rendering, there are certain issues that can occur.

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 [4, 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.

2.4.2 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 with *vertex morphing* [1, 7, 9], 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" [11] by Lindstrom *et al.* in 1996.
- **"ROAMing Terrain: Real-time Optimally Adapting Meshes"** [5] by Duchaineau *et al.* in 1997.
- "Real-Time Generation of Continuous Levels of Detail for Height Fields" [6] by Röttger *et al.* in 1998.
- **"Fast Terrain Rendering Using Geometrical MipMapping"** [1] by de Boer in 2000.
- "Rendering Massive Terrains using Chunked Level of Detail Control" [8] by Ulrich in 2002.
- "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids" [7] by Hoppe and Losasso in 2004 and the follow-up **"Terrain Rendering Using GPU-Based Geometry Clipmaps"** by Asirvatham and Hoppe [2] in 2005.
- "Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD)" [9] by Strugar in 2009.

- **"Concurrent Binary Trees (with application to longest edge bisection)"** [10] 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.* [5] published in 1997. ROAM represents the terrain mesh using bintrees and performs triangle splits and merges for generating and removing detail.

The central idea of the algorithm is to use temporal coherence: often, between two frames the meshes are 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 distance between where a node is supposed to be on the screen and where the algorithm actually placed the node. The maximum of all distances is calculated and used as the base priority metric of the algorithm.

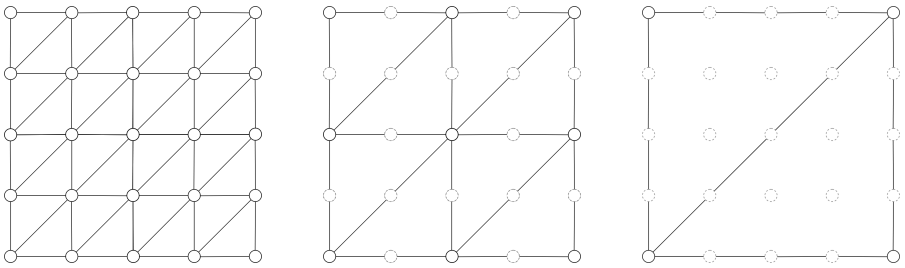
While the bintree trees gets traversed, various flags are updated which indicate whether a wedgie is inside the view-frustum, partially or not at all. ROAM is designed as a greedy algorithm, meaning it will always performs the most optimal splits/merges for each frame.

3.1.2 GeoMipMapping

Geometrical Mipmapping (GeoMipMapping) is a terrain LOD approach published by de Boer [1] 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[1] $0 \leq l \leq n$ 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 3.1 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 3.1: Example of each GeoMipMap of a 5×5 block. The omitted vertices of lower LOD GeoMipMaps are marked as dotted circles (based on [1]).

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 3.2.

[1]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.

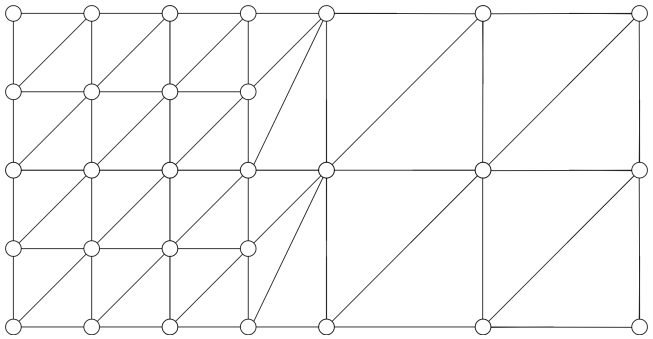


Figure 3.2: Example of GeoMipMapping's crack avoidance between a LOD 2 and a LOD 1 GeoMipMap of two 5×5 blocks (based on [1]).

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 [7] is a terrain rendering technique published by Hoppe and Losasso in 2004. A follow-up GPU-based variant of Geometry Clipmaps [2] 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 3.3 shows an example of this mesh.

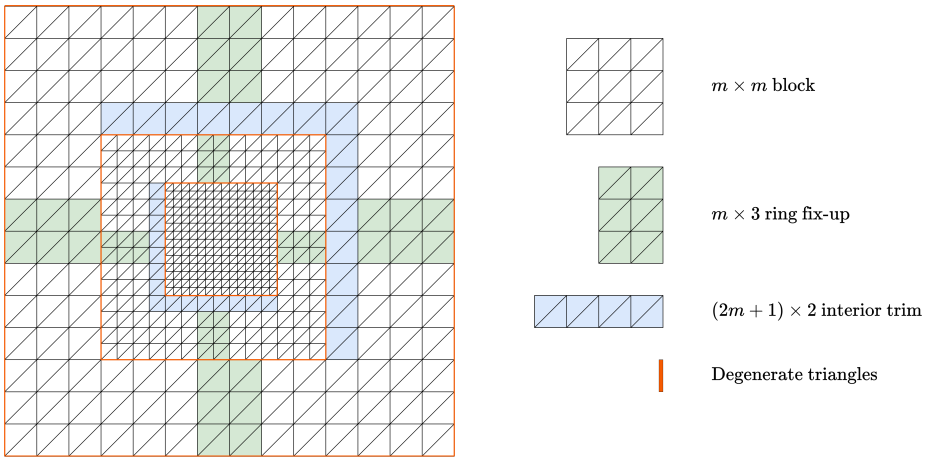


Figure 3.3: Example of the flat mesh in Geometry Clipmaps with $n = 15$, $m = 4$ and $l = 3$ (based on [2]).

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 viewer 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, which then linearly grows 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. The normal map is computed whenever the clipmap gets updated.

3.1.4 Concurrent Binary Trees

The *concurrent binary tree (CBT)* is a data structure published by Dupuy in 2020. It essentially allows for binary trees to be computed in parallel using a binary heap stored 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 describing the application of CBTs to terrain rendering. The approach is similar to [5] 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 happens in parallel on compute shaders with the CBT data structure, whereas in [5], the bintrees are split and merged on the CPU. The split and merge criteria are defined such that sub-pixel rasterization is avoided. Triangles outside of the view-frustum and triangles at flat areas are not split or merged further.

Dupuy claims

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 [7], 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.

GeoMipMapping The strongest point of GeoMipMapping is the fact that its easy to understand and to implement. GeoMipMapping was published at a time in which immediate mode was widely-used. Nowadays, mmediate mode is deprecated in most graphics APIs. In order for GeoMipMapping to be suitable for modern GPUs, it needs to be modified such that it can work with vertex and index buffers, which is feasible thanks to its block-based nature.

Geometry Clipmaps GPU-based Geometry Clipmaps was one of the first algorithms to utilize the vertex texturing functionality of GPU, which was newly introduced at the time of its publication. The fact that only very few vertices and indices are required (which are translated and scaled during rendering to their corresponding position) and the fact that the heightmap can be sampled in the vertex shader makes GPU-based Geometry Clipmaps still a suitable approach for modern hardware. This is reflected in the fact that the most widely-used Godot plugin for heightmap rendering is based on GPU-based Geometry Clipmaps, (see the next section). Some other strong points are the transition regions for avoiding pops, its configurability, and the fact that no LOD determination has to be performed, since the mesh is constant.

CBT The CBT data structure "revitalized" mesh-subdivision-based approaches, such as [11, 5], since bintrees can now be computed in parallel with compute shaders, rather than on the CPU. However, some aspects are not yet handled, such as preventing pops. It is a rather new approach that has not yet been tested in the real-world.

Overall Conclusion Overall, algorithms which load the vertices and indices once to the GPU and do not touch the buffers again

3.2 Terrain LOD in Real-world Systems

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 [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.

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 is given and some additional information on related projects.

The terrain system supports importing and exporting of heightmaps in the RAW file format, with either an 8-bit or 16-bit grayscale value per pixel. Visually, the mesh of the terrain LOD resembles that from a quadtree-based LOD approach, such as [8]. 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 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 developed by TODO that utilizes the fast quadratic mesh simplification algorithm developed by Sven Forstmann TODO citation.

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. As of today, Unity does not yet use the CBTs for its 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.

The maximum supported heightmap size is 8192×8192.

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. 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.

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 Preliminaries

4.1.1 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.
- ImGui: *Dear ImGui* is a multi-platform graphical user interface library developed by Omar Cornut TODO cite.
- 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.

4.2 Basic Setup and Architecture

4.2.1 Overview

TODO High-level class diagram

4.2.2 Shaders

The class `Shader` encapsulates a shader program consisting of a vertex shader and a fragment shader. It's based on the "Shaders" chapter in *Learn OpenGL - Graphics Programming* [15].

4.2.3 Camera

The camera class is based on the "Camera" chapter in *Learn OpenGL - Graphics Programming* [15]. The camera is defined by its yaw and pitch angles, which are used to construct the front, up and right camera vectors. The camera also consists of its position, of its near and far z-values, of its aspect ratio, zoom, and of its flight and look-around velocities. The aspect ratio, near and far values and zoom are used to

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 **??**

```
float posLerp = 0.0f;
// ...
void run() {
    while (!glfwWindowShouldClose(window)) {
        // ...
        if (camera.isFlying) {
            camera.lerpFly(posLerp);
            posLerp += 0.0005 + flightVel / 50000;

            if (posLerp >= 1.0f) {
                camera.isFlying = false;
                posLerp = 0.0f;
            }
        }
        // ...
    }
}
```

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

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

4.2.4 Skybox

A *skybox* is a box in world space that simulates the sky using six texture images, one for each side of the box. Skyboxes are rendered with *cubemaps*

The skybox implemented in ATLOD is based on the "Cubemaps" section in the "Advanced OpenGL" chapter in *Learn OpenGL - Graphics Programming* [15]. It is encapsulated in the class `Skybox`.

An improvement over the current skybox would be to actually calculate the *atmospheric scattering*, which would deliver a more realistic and flexible daytime-based lighting. A suitable approach would be *precomputed atmospheric scattering* by Bruneton and Neyret [16].

4.2.5 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, such as the heightmap, shader, width, height, and more. It also contains the three virtual methods `loadBuffers`, `render()` and `unloadBuffers()`, which all terrain subclasses must implement.

The base terrain is structured as shown in listing TODO.

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.

Heightmap Preprocessing

Digital elevation model (DEM) data is commonly offered in the GeoTIFF file format by various DEM providers, such as SwissTopo and OpenTopography. GeoTIFF files can be converted into PNG using GIS software, such as QGIS or GDAL.

A small Python 3 command line utility named `geotiff-to-png` for converting GeoTIFF files into 16-bit grayscale PNG images is available in the repository of ATLOD in the `scripts` folder.

Loading

The method `load()` is responsible for loading a heightmap located at a given file path `fileName`. Depending on the file extension of `fileName`, TODO.

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 if needed.

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 4.1 shows the organization of indices for rendering the terrain as triangle strips.

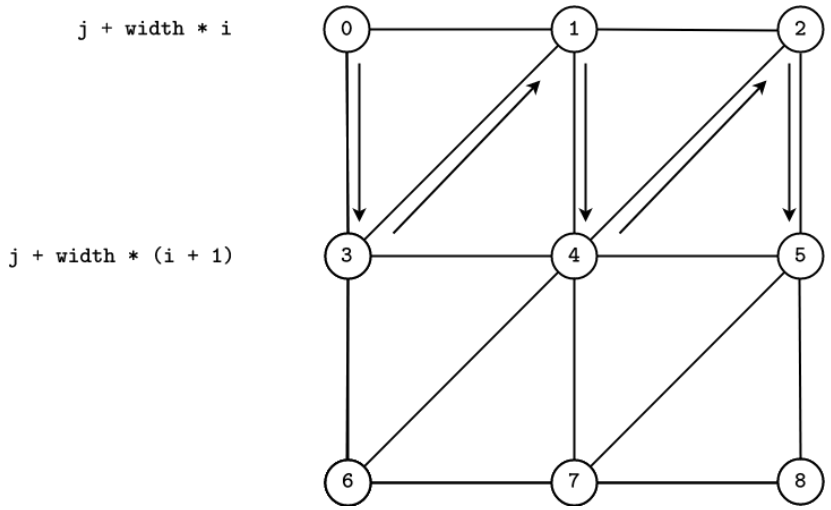


Figure 4.1: Example of a terrain layout for triangle strips. The looping index `i` goes from 0 to the terrain height and `j` from 0 to the terrain width. The final indices to be rendered are 0,3,1,4,2,5,`RESTART`,3,6,4,7,5,8,`RESTART`.

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 are calculated as follows. Each vertex consists of its $\mathbf{p}_{pos} = (x, y, z)$ position, where the (x, z)-coordinates are given by the current position relative to the heightmap size and the y-coordinate is retrieved from the heightmap in memory. The four orhogonally neighboring points

Geometrically, the direction of a sum of vectors is given by constructing a chain of these vectors, such that the beginning of the next vector is at the tip of the previous vector, and then "walking" along all vectors from the first to the last vector.

4.3.2 Shading

The shading is calculated with the Phong shading technique in the fragment shader.

4.4 GeoMipMapping

The implementation supports most basic functionalities described in the original paper, but differs in a few key aspects. It also draws some inspiration from other approaches, most notably from GPU-based Geometry Clipmaps by Hoppe and Asirvatham and from "Terrain Rendering in Frostbite Using Procedural Shader Splatting" by Andersson for the Frostbite game engine TODO cite. Both approaches utilize a single flat mesh (positioned around the viewer in Hoppe and

Asirvatham's approach) 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 object 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, which is stored as a texture object. The upcoming subsections describe the approach in greater detail.

4.4.1 Block Structure

As 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 the structure containing information of a particular section of the terrain. The class `GeoMipMappingBlock` represents such a block and contains the following fields:

- `unsigned _blockId`: this field stores the ID of the current block.
- `unsigned _currentBorderBitmap`: this field stores the current border permutation as a bitmap.
- `float _minY` and `float _maxY`: these two fields store the minimum and maximum y-coordinates of that particular block.
- `glm::vec2 _translation`: this field stores the 2-dimensional translation vector for translating the flat mesh to the block's actual position.
- `glm::vec3 _aabbCenter` and `glm::vec3 _trueCenter`: `_aabbCenter` stores the position of the center of the AABB of the block, i.e. its y-coordinate is set to `_maxY` − `_minY`. This field is used for view-frustum culling during rendering. The field `_trueCenter` on the other hand stores 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.

4.4.2 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 $b \times b$ mesh centered around $(0, 0, 0)$, where $b = 2^n + 1$ is the block size 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 explain in the next section. Figure 4.2 shows the described index buffer organisation.

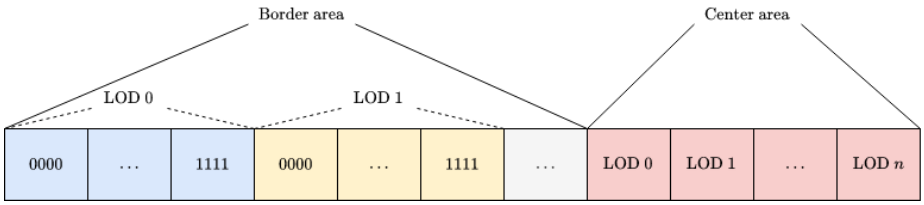
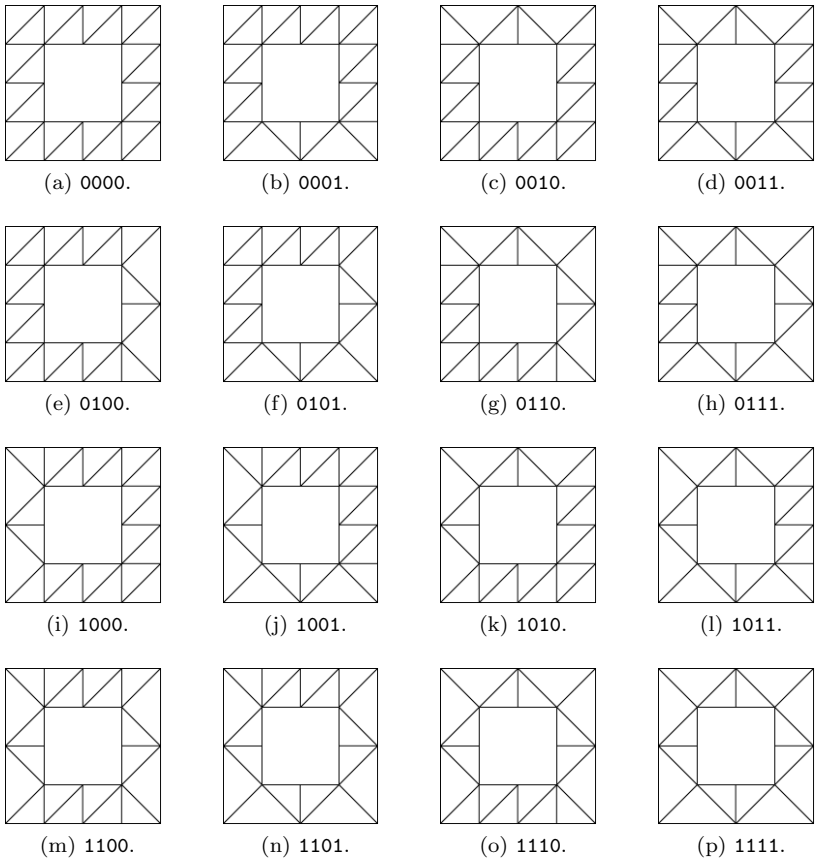


Figure 4.2: The index buffer organisation of the single flat block. The variable n corresponds to the maximum LOD level.

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. A similar approach is described by Andersson in SIGGRAPH 2007 for the terrain rendering in the Frostbite game engine for the game Battlefield 2, with their approach requiring only 9 permutations. Figure 4.3 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 4.3: Every possible border permutation for a LOD 2 GeoMipMap of a 5×5 block. The center subblocks have been omitted from the illustration.

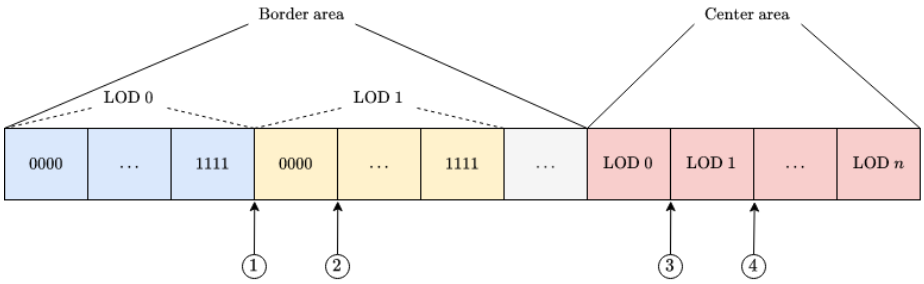
This makes clear why the flat mesh is split into its border and center area. The center area only depends on the LOD level and is the same regardless of the current border permutation. Not splitting the flat mesh up into its border and center area would require longer index buffer generation times and 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. 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 4.4 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 4.4: 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()` 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 4.1: 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 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 as borders and do not have a center.
- It loads the rest of the indices from LOD 2 to the maximum LOD level, by first loading in the border indices with `loadBorderAreaForLod()` and then the center areas with `loadCenterAreaForLod()`.

4.4.3 LOD Selection

The LOD of ATLOD's GeoMipMapping implementation is based on the Euclidean distance *dist* between the camera's position $\mathbf{p}_{camPos} = (camPos_x, camPos_y, camPos_z)$

and the block's center point $\mathbf{p}_{blockCenter} = (blockCenter_x, blockCenter_y, blockCenter_z)$

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

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:

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

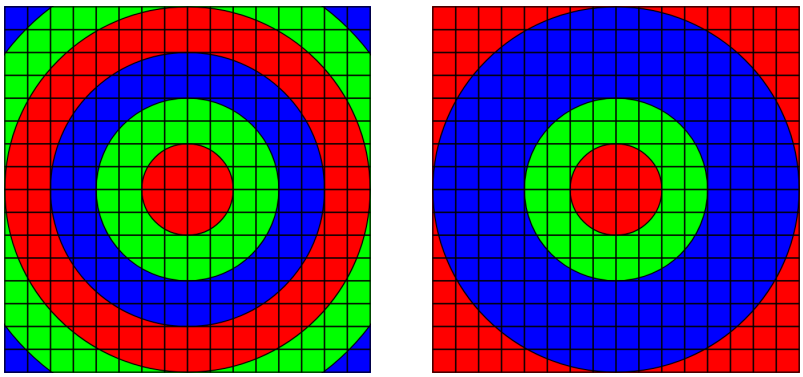
The above expressions are implemented in the `GeoMipMapping` class in a single method `determineLodDistance()`. The LOD determination mode can be set with the boolean argument `doubleEachLevel`. Additionally, the minimum and maximum LOD levels can be manually set to be different from 0 and l respectively 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 4.2 shows the method `determineLodDistance`.

```
1 unsigned GeoMipMapping::determineLodDistance(float
      squareDistance, float baseDist, bool doubleEachLevel)
2 {
3     unsigned distancePower = 1;
4     for (unsigned i = 0; i < _maxLod - _minLod; i++) {
5         if (squaredDistance < distancePower * distancePower
      * baseDist * baseDist)
```

```
6                 return _maxLod - i;
7
8             if (doubleEachLevel)
9                 distancePower <<= 1;
10            else
11                distancePower++;
12        }
13        return _minLod;
14    }
```

Listing 4.2: Method `GeoMipMapping::determineLodDistance()` that determines the LOD level of a block based on its distance to the camera.

Figure 4.5 shows both LOD determination modes in action.



(a) Linearly growing distance mode.

(b) Exponentially growing distance mode.

Figure 4.5: 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.

4.4.4 View-frustum Culling

View-frustum culling is implemented with the methods TODO in the `Camera` class.

4.4.5 Rendering

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

- The first phase iterates through every block, calculates the distance between the block's `_trueCenter` and the camera's position, and updates the LOD level of the block accordingly.
- The second phase consists of performing view-frustum culling, setting the uniform variables in the shader, and finally rendering the block.

Vertex Shader

The vertex shader first translates the vertex from its initial position around $(0, 0, 0)$ to its actual world-space position using the uniform `vec2` variable `translation`.

Fragment Shader

The fragment shader

The Phong shading is based on TODO cite and is performed with the following steps: first, the heightmap is sampled at the four orthogonally neighboring points $\mathbf{p}_{1,0} = (x+1, z)$, $\mathbf{p}_{-1,0} = (x-1, z)$, $\mathbf{p}_{0,1} = (x, z+1)$ and $\mathbf{p}_{0,-1} = (x, z-1)$, where (x, z) is the current texture coordinate for sampling the current height from the heightmap. Using these four points, the slope in x and z-direction can be calculated by computing

$$dx = \mathbf{p}_{-1,0} - \mathbf{p}_{1,0}$$
$$dz = \mathbf{p}_{0,-1} - \mathbf{p}_{0,1}.$$

These values can now be used to create a normal vector

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

Listing TODO shows the calculation of the normal vector from the heightmap texture.

4.4.6 Conclusion

The implementation still has some room for improvements, such as instanced rendering, geomorphing and a quadtree-based frustum culling. The memory consumption by the index buffer could be further brought down by only defining indices for the border permutations $(0, 0, 0, 0)$, $(1, 1, 1, 1)$, $(1, 1, 0, 0)$, $(1, 0, 1, 0)$,$(1, 1, 1, 0)$ and by rotating the mesh about the y-axis such that the borders align without cracks.

Chapter 5

Results

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

5.1 Experimental Setup

5.1.1 Computer

The computer 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

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 TODO cite 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. The heightmap file is a 13922×14140 16-bit greyscale PNG image converted from a GeoTIFF file, as shown in figure 5.1.

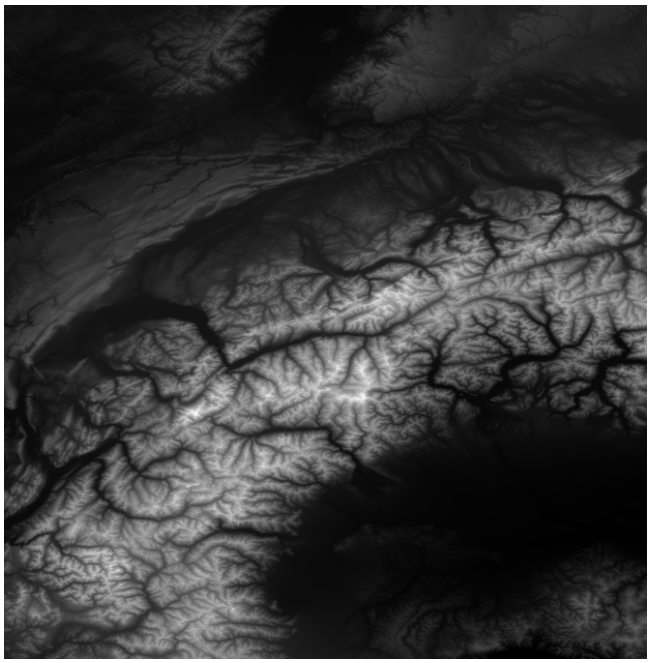


Figure 5.1: The 13922×14140 16-bit greyscale heightmap used for benchmarking (retrieved from OpenTopography TODO cite). In this figure, the gray values were converted from 0,...,65535 to 0,...,255 in order to make the heights more visible.

The GeoMipMapping algorithm is configured for the best balance between performance and visual accuracy as follows:

- Block size: 257
- Fog: 0.003
- Minimum LOD: 0
- Maximum LOD: 8
- Base distance: 700
- LOD determination mode: Exponentially growing distance

5.1.3 Benchmarks

Two kinds of benchmarks are performed:

- The first benchmark measures the performance, which measures the average FPS during rendering.
- The second benchmark measures the visual accuracy, i.e. the image difference between the ground truth (actual full-resolution terrain) and the terrain rendered with LOD.

For each of the benchmarks, 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.

5.2 Performance

5.2.1 Flyover from Corner to Corner

5.2.2 360° Rotation

5.3 Acurracy

5.3.1 Flyover from Corner to Corner

5.3.2 360° Rotation

5.4 Theoretical Memory Consumption

5.4.1 RAM

5.4.2 GPU Memory

While there is some complexity overhead in managing the index buffer in the presented way, the GPU memory usage is quite low. The main bottleneck of this implementation in terms of GPU memory is the heightmap texture, which takes up 2 bytes per height value. A solution 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.

On the other hand, memory consumption by the vertices and indices is quite low. The number of vertices that are loaded on the GPU is only $b \times b$.

Table TODO shows memory usage by the heightmap texture, the vertex buffer and the index buffer with certain terrain sizes and block sizes.

Chapter 6

Discussion and Conclusion

6.1 Further Work

ATLOD can be extended in a number of ways:

- Additional terrain LOD algorithms: ATLOD currently only supports GeoMipMapping. Some interesting and relevant algorithms that could be added are GPU-based Geometry Clipmaps, CDLOD and Concurrent Binary Trees.
- Terrain streaming/paging: currently, ATLOD only loads a single instance of `GeoMipMapping`. ATLOD could be extended to load multple instances with dynamic loading and offloading of instances and heightmaps, which would allow for even larger (theoretically infinite) terrains.

6.2 Reflexion

Bibliography

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

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

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

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

[5] 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.

[6] 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.

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

[9] 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.

[10] Jonathan Dupuy. Concurrent binary trees (with application to longest edge bisection). *Proc. ACM Comput. Graph. Interact. Tech.*, 3(2), aug 2020. doi:10.1145/3406186.

[11] 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.

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

[16] 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. doi:10.1111/j.1467-8659.2008.01245.x.