Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Implementation of a Real-time Streaming-based Terrain Level of Detail System

Bachelor Thesis by

Amar Tabakovic

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

Supervisor
Prof. Marcus Hudritsch

External expert
Dr. Eric Dubuis

June 13, 2024

Abstract

Terrains are an important part of various practical applications of computer graphics, such as video games, flight simulators and geographical information systems. Since terrains are expensive to render, special rendering optimizations, such as level of detail (LOD). Besides the rendering performance, another important aspect to terrain rendering is the management of terrain data. Terrain datasets that are too large to fit entirely in memory must be streamed in from the disk or over the network and streamed out of memory.

This thesis describes the implementation of StreamingATLOD, a large-scale terrain rendering system with LOD that is capable of rendering the Earth by streaming in height data and satellite imagery from web APIs based on the XYZ tiling scheme. StreamingATLOD is currently built around the data and APIs from the company MapTiler, which serve height data tiles and satellite imagery tiles. Both layers come projected in the Web Mercator projection and are used up to zoom level 14, which corresponds to a precision of 9.5 meters per pixel.

The implemented terrain LOD algorithm is mainly based on Chunked LOD. The terrain is organized as a quadtree, where each quadtree node represents a section of the terrain at the resolution of the node's depth. Each node gets rendered with heightmap-displacement in the vertex shader followed by a projection onto the globe. In order to hide cracks between adjacent nodes, skirts are rendered. The Web Mercator projection cuts off at a latitude of 85.05°, resulting in holes at the North and South Pole. These holes get covered with a circular pole mesh. Terrain sections which are not visible are culled with view-frustum culling and horizon culling.

The terrain data is cached in a memory cache and a disk cache. Both caches are least-recently used (LRU) caches, which automatically evict the least-recently used data upon reaching the maximum capacity, thus keeping only the most necessary terrain data in memory and on the disk respectively. New terrain data is streamed in with separate load worker threads, allowing the render thread to continue rendering while data is loaded in the background. The load worker threads load data either from the disk cache or from the web API. Disk cache entries that need to be evicted are deleted from the disk using a separate disk deallocation thread.

StreamingATLOD's rendering and streaming performance were benchmarked on a 2020 Intel MacBook Air with the disk caching disabled. StreamingATLOD yielded on average around 60 frames per second and was able to load the data required for rendering at a reasonable speed.

Some technical aspects in which StreamingATLOD could be improved are prioritized loading of terrain data, better high precision support and rendering transitions between LOD level changes. Possible extensions and future work include incorporating a realistic atmosphere, supporting multiple tile layers and extending StreamingATLOD into a software development kit.

Contents

List of Tables

List of Figures

List of Listings

Preface

Personal Motivation

Why terrain rendering? It all started during the fourth semester of my studies, where I had to choose a topic for the preceeding project course "Project 2". I saw in the list of project proposals that terrain level of detail was one of the offered topics and after a first quick research on the topic, I was hooked immediately, not only because I found terrain rendering interesting and visually stunning, but mainly because I had the opportunity to challenge myself with a (subjectively) difficult topic. As part of the preceeding project, I implemented a small terrain renderer with LOD which worked decently and served as a good base for further work and exploration. But I wanted to render the probably most extreme and well-known example of a terrain there is: the entire Earth. This of course meant tackling entirely new challenges, such as geographic projections, multithreading, streaming and caching of terrain data, but this is exactly what made this project more exciting. In the end, the opportunity of working a project like this does not come every day.

Acknowledegments

I would like to thank Prof. Marcus Hudritsch for the opportunity to write this thesis and for our the pleasant cooperation. I always enjoyed our weekly meetings and your support. I would also like to thank Dr. Eric Dubuis for supervising this thesis and for his crucial inputs during our intermediate meeting. Also thanks to Marino for joining the weekly meetings towards the end of the thesis.

I would finally like to thank my family and my girlfriend Cristina for supporting me throughout my studies.

Chapter 1

Introduction

Terrains are an important part of many practical applications of 3D computer graphics. They can be found in video games, simulation software and geographical information systems (GIS). Rendering terrains, however, is far from an easy task and there are numerous issues which need to be addressed.

The first main issue is that terrains are expensive to render due to their near-constant visibility and their sheer size. For example, consider a square terrain which consists of 8192×8192 vertices. Rendering the entire terrain without any optimizations would require more than 134 million triangles[1] to be rendered per frame, which is completely infeasible even with the most performant GPUs of today. The solution for this issue is *level of detail (LOD)*: sections of the terrain which are far away from the camera or with low variation in height are rendered with fewer vertices and triangles. Terrain LOD has been the topic of considerable research throughout the last three decades, spawning numerous algorithms and approaches which solve this issue in a variety of creative ways.

The second main issue concerns the management of terrain data. Terrain datasets can become extremely large, especially when representing real places, such as the Earth. For example, in 2016 the Google Earth database was estimated to contain more than 3000 terabytes of data in total [Whi]. Most consumer computers today have between 8 to 32 gigabytes of RAM and 2 to 6 gigabytes of GPU memory, making loading the entire dataset into memory impossible. The solution for this issue is *streaming*: instead of loading the entire terrain data all at once into memory, the idea is to only *stream in* the currently needed data from the disk or over the network. Over time, unused terrain data must be deallocated as well in order to make space for new data.

When terrain streaming is performed over the network, a problem that can arise is that the the terrain data servers get overwhelmed with lots of requests. The solution for this problem is *caching*: the concept of temporarily keeping recently or frequently used terrain data on the client in order to reduce the number of network requests. Terrain caching is usually performed in multiple layers. Terrain data can be situated (in the order of accessing speed): on the GPU,

[1] Each vertex represents one height point and each quad consisting of 4 vertices consists of two triangles. The total number of triangles is given by $(8192-1) \times (8192-1) \times 2 = 134184962$.

in memory, on the disk or on the terrain server [CR11, p. 382]. Both terrain streaming and caching draw inspiration mainly from the areas of operating systems, web browsers and data management.

1.1 Goals of this Thesis

The main goal of this thesis is the development of a prototype for a streaming-based terrain rendering system with level of detail. The system should meet the following requirements.

Terrain Rendering The system should be capable of rendering a terrain at interactive framerates using level of detail and other optimizations to ensure appropriate rendering performance. Preferably, the terrain should be rendered as a sphere for better realism.

Terrain Streaming The system should be capable of loading and offloading terrain data in and out of memory based on the camera's state. Preferably, the terrain data can be loaded at runtime from web-based APIs served by data providers.

Terrain Caching The system should be capable of caching terrain data in order to avoid unnecessary requests for recently or frequently used data.

Additional Features Some additional, but not crucial, features include proper camera handling, collision detection, atmospheric rendering and support for a configuration file for adjusting various parameters.

1.1.1 Restrictions

The following aspects were explicitly left out of scope of this thesis.

Rigorous Testing and Deployment The developed system serves as a first proof-of-concept since this thesis focuses mainly on the development of the features. For future further development after this thesis, rigorous testing and deployment is indespensable.

Development and Deployment of Custom Terrain Data Servers This thesis does not include the development and deployment of custom terrain data servers. The focus of this thesis is on the client-side rather than on the server-side.

Vector Tiles, 3D Buildings and Other Geospatial Features This thesis focuses mainly on terrains. Incorporating other geospatial features, such as vector tiles, 3D buildings, points-of-interests, etc. would not be realistic in such a short timespan.

1.2 Intended Readership

The reader is assumed to be familiar with the basics of computer science, C++ programming and 3D computer graphics. The main concepts of terrain rendering will be introduced when required in the subsequent chapters.

1.3 Notation and Terminology

This thesis uses the following mathematical notation:

- By default, the coordinate system is a right-handed coordinate system with y as the up-direction, unless explicitly stated otherwise.
- The set of natural numbers is denoted $\mathbb{N}$ and the set of real numbers is denoted $\mathbb{R}$.
- Points in $\mathbb{R}^3$ are denoted $\mathbf{p} = (p_x, p_y, p_z)$.
- Vectors in $\mathbb{R}^3$ are denoted $\mathbf{v} = (v_x, v_y, v_z)$.
- Matrices in $\mathbb{R}^{n \times n}$ are denoted $\mathbf{M}$ and are column-major.
- XYZ tile keys (introduced in chapter "Theoretical Background") are denoted $tk = (tk_x, tk_y, tk_z)$.
- Geodetic coordinates (introduced in chapter "Theoretical Background") are denoted $\mathbf{p}_{geodetic} = (lon, lat, h)$.
- The vector containing the globe radii for the geodetic-to-cartesian transformation (introduced in chapter "Theoretical Background") is denoted $\mathbf{r}_{ellipsoid} = (r_x, r_y, r_z)$.

1.4 Structure of this Thesis

This thesis is structured as follows:

- Chapter 2 introduces the reader to various topics covered in this thesis, such as the basics of terrain rendering and some concepts from geographical information systems.
- Chapter 3 provides a short overview of previous work conducted in the area of real-time terrain rendering and streaming and gives a short recapitulation of the author's preceeding project "3D Terrain with Level of Detail".
- Chapter 4 goes into the implementation details of *StreamingATLOD*, the implemented streaming-based terrain renderer. First, some preliminary information is given, such as the used data and technologies. Afterwards, various aspects of StreamingATLOD are described in multiple sections.
- Chapter 5 describes the results, including a selection of screenshots of the Earth and various performance benchmarks.
- Chapter 6 gives a discussion of the obtained results.

- Chapter 7 concludes this thesis with a summary and some potential improvements and future work.
- Appendix A contains the time schedule for this thesis.

Chapter 2

Theoretical Background

This chapter introduces some background knowledge related to terrain rendering and geographical information systems (GIS).

2.1 Basics of Terrain Rendering

2.1.1 Terrain Data Representations

One of the central aspects of terrain rendering is the underlying data representing the terrain. The following subsections introduce the two most important ways of representing terrain data and lists some of their strengths and weaknesses.

Heightmaps

The surface of a terrain can be represented with a regular grid of height values. When storing such height values in an image, the resulting image is called a *heightmap*. Each pixel in the heightmap contains a color value, which corresponds to the height of the terrain at that particular image coordinate. Normally, low color values correspond to low elevation and high color values to high elevation. Figure 2.1 shows an example of a 16-bit greyscale heightmap of a large part of Switzerland and its neighboring countries.

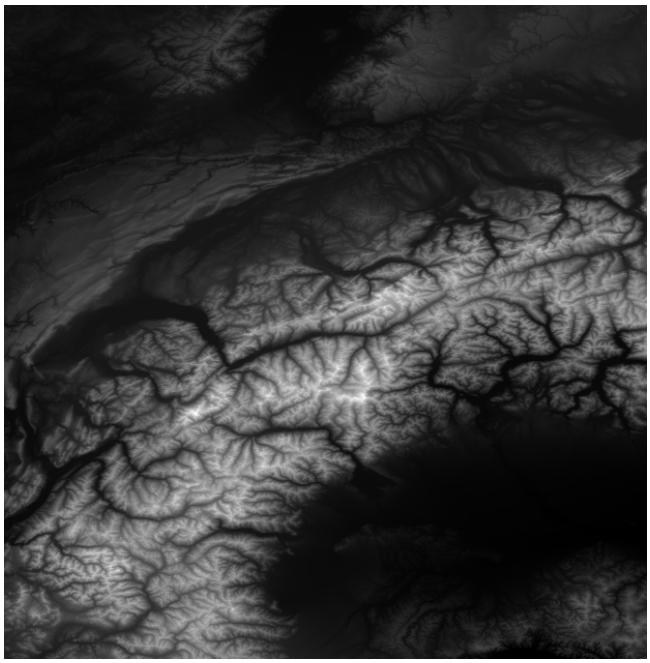


Figure 2.1: The 13922 × 14140 16-bit greyscale heightmap of a large part of Switzerland and neighboring countries (retrieved from OpenTopography [NAS13]). In this figure, the gray values were converted from 0,...,65535 to 0,...,255 in order to make the heights more visible.

The main advantage of heightmaps is that height data can be stored and retrieved very efficiently, requiring only the actual height value to be stored per position instead of the entire (x, y, z) coordinate. Another advantage of heightmaps is that they can be used as textures on the GPU, which has been basis for many modern GPU-based terrain LOD algorithms (see subsection "Categorization of Terrain LOD Algorithms").

The main weakness of heightmaps is that they are limited to a single height value per (x, z) coordinate. Special terrain features such as cliffs, overhangs or caves cannot be modelled only with heightmaps and require special handling [LWC+02].

Heightmaps are used extensively in game engines. The Unity game engine for example stores heightmaps as 16-bit-per-color grayscale RAW images, allowing for $2^{16} = 65536$ different height values.

Triangulated Irregular Networks

An alternative representation of a terrain's surface is the *triangulated irregular network (TIN)*. The TIN is a data structure that consists of 3-dimensional points which make up a triangulated mesh. This means that when viewed from above (i.e. from the (x, z) plane), the distance between any two neighboring points can be irregular, which is not the case with heightmaps. TINs are often generated using heightmaps themselves for height information.

Figure 2.2 shows two examples of TINs; a TIN with a highly variable surface and a TIN with a less variable surface.

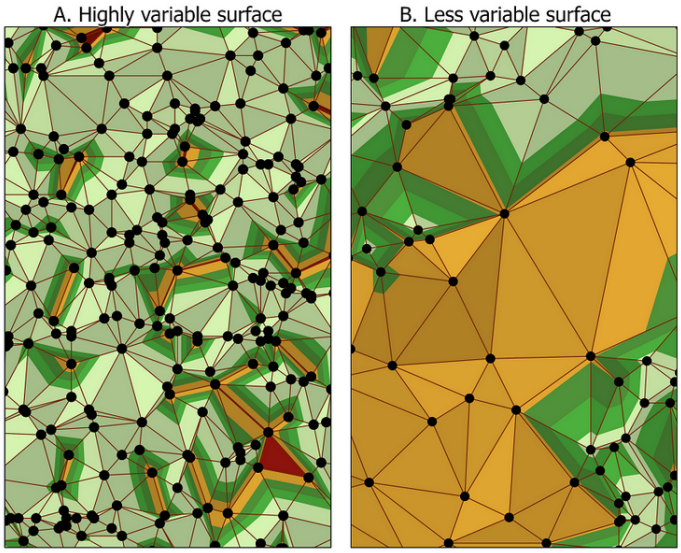


Figure 2.2: Two examples of TINs: a TIN with a highly variable surface (A) and a TIN with a less variable surface (B) (image source: [UCfGIS]).

The main advantage of TINs is that unnecessary information in the terrain can be eliminated. For instance, flat areas with little variation in height can be represented with few vertices. Another advantage is that special terrain features, such as cliffs or overhangs, can be modelled easily, since each point is a three-dimensional vertex.

A disadvantage of TINs is that they are more expensive to compute and prepare, especially when taking LOD into account. Another disadvantage of TINs is that they now require the full (x, y, z) vertices to be stored per point in the TIN, whereas with heightmaps, only a single color value per height is required.

2.1.2 Categorization of Terrain LOD Approaches

The following categorization of terrain LOD algorithms is partially based on chapter "Massive-Terrain Rendering" from the book *3D Engine Design for Virtual Globes* by Ring and Cozzi [CR11, p. 365].

Discrete LOD *Discrete LOD* algorithms usually work on discrete blocks or patches of terrain. These blocks are usually organized into multiple LOD resolutions. Some examples of discrete LOD algorithms include GeoMipMapping [dB00] and Geometry Clipmaps [LH04].

Continuous LOD *Continuous LOD* algorithms usually operate on single triangles and vertices and dynamically modify the mesh at runtime. Continuous LOD algorithms are not widely used today, since modifying the mesh on a

frame-to-frame basis at runtime is less performant than the alternative of simply storing terrain blocks on the GPU in advance[CR11, p. 368]. For this reason, this thesis will not continue discussing continuous LOD algorithms. Some examples of continuous LOD algorithms include *ROAM* [DWS+97] and *SOAR* [LP01].

Hierarchical LOD *Hierarchical LOD* algorithms are similar to discrete LOD algorithms in the sense that they organize the terrain in discrete blocks. The main difference is that hierarchical LOD algorithms organize the blocks hierarchically so that the data further up in the hierarchy represents larger areas at lower resolutions and the data lower in the hierarchy represents more detailed areas at higher resolutions. The actual mesh can be either grid-based, as in GeoMipMapping or Geometry Clipmaps, but it can also be based on TINs. The most well-known example of a hierarchical LOD algorithm is Chunked LOD [Ulr02].

A small selection of terrain LOD algorithms relevant for this thesis will be summarized in chapter "Previous Work".

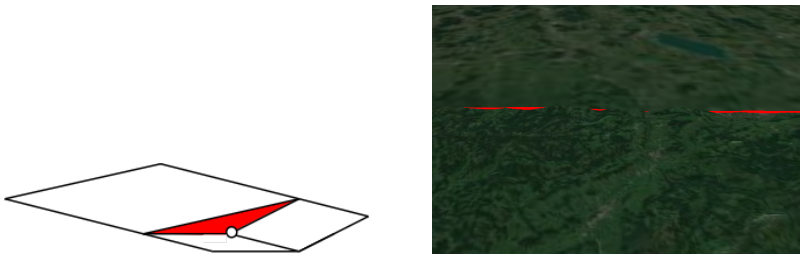
2.1.3 Potential Problems during Rendering

Low Framerates

As already indicated in the introduction chapter, the most prevailing problem during terrain rendering and the main motivation behind terrain LOD algorithms is a bad rendering performance due to the massive number of rendered triangles. Systems that render terrains must be able to do so with decent framerates of at least 60 frames per second. A low framerate destroys immersion of the user and puts a major strain on the GPU.

Cracks

When rendering terrains with LOD, a common scenario is that a terrain region close to the camera with more detail borders another region of terrain further away with less detail. Without any special treatment, this scenario can cause a visual disturbance in the terrain, a so-called *crack*. Cracks occur when certain vertices of the higher resolution terrain mesh lie below the edge of the lower resolution terrain mesh, as illustrated in figure 2.3.



(a) (b)

Figure 2.3: An illustration (a) and a render (b) of a crack occuring between two terrain sections with different LOD levels. The background was colored red in both subfigures for visibility.

Cracks can be avoided in a number of ways. One approach is to organize the terrain meshes in such a way that such crack-causing vertices can never occur in the first place. This approach is used in numerous terrain LOD algorithms, such as GeoMipMapping [dB00], ROAM [DWS+97] and others. The main advantage of this approach is that it ensures a continuous terrain mesh. The disadvantage is that it requires potentially more complex mesh processing and triangulation times.

Another approach is to render *skirts* around the terrain meshes, which drape downwards and repeat the overlay texture at the border. This approach is used in various terrain LOD algorithms as well, such as Chunked LOD [Ulr02], Geometry Clipmaps [LH04] and others. The main advantage of skirts is that they are very simple to implement. The main disadvantage is that they require more triangles to be rendered and are thus slightly more expensive to render. Another disadvantage is that skirts are only hiding the cracks instead of completely removing them.

Popping

Popping is a visual artifact that occurs when the camera is flying towards a lower resolution terrain section and the sudden change in LOD level of the terrain section to a higher LOD level causes a visual popping-in of more detail into the screen.

Popping can be avoided in a number of ways. The first solution is *geomorphing* [Wik24a]. With geomorphing, the vertices of the higher and lower resolution terrain meshes are interpolated, such that a smooth transition occurs during when flying closer to the target terrain mesh. Another solution is *LOD blending* [Wik24a], which simply blends the alpha or transparency channel of both the high and the low resolution mesh of the same terrain section, at the cost of having to render both meshes at the same time.

Precision Issues

A set of common issues for particularly large and detailed terrains are *precision issues*. Most graphics APIs today rely on 32-bit floating point numbers by

default. When using 32-bit floating point numbers for transforming vertices with very large coordinates, this can lead to *jittering*, i.e. vertices shaking on the screen due to roundoff errors. The depth buffer also suffers from limited precision (24 bits in OpenGL by default), which may lead to *depth buffer fighting*, where objects on the screen which are close to each other flicker repeatedly due to roundoff errors in the depth buffer.

The reference book *3D Engine Design for Virtual Globes* by Cozzi and Ring lists a few common solutions for handling precision issues for both vertex transformation imprecision and depth buffer imprecision. Vertex transformation imprecision artifacts can be avoided by using OpenGL's double precision support (when supported), *relative-to-center rendering* [CR11, p. 164], *relative-to-eye rendering* [CR11, p. 169], or when applicable, scaling down coordinates. Depth buffer imprecision artifacts can be avoided with *complementary depth buffering* [CR11, p. 189], *logarithmic depth buffering* [CR11, p. 191] and *rendering using multiple view-frustums* [CR11, p. 194].

2.2 Digital Cartography

This section introduces a selection of concepts from geographical information systems which will later be important for accurately rendering the Earth in StreamingATLOD.

2.2.1 Ellipsoids

The Earth, despite vocal objections by many throughout the course of history, is not flat. Contrary to popular belief, the Earth is not entirely spherical either, but *ellipsoidal*. This means that the radius of the Earth is not the same in the (x, y, z) directions. An ellipsoid is defined with three radii $\mathbf{r}_{ellipsoid} = (r_x, r_y, r_c)$ and each point (x, y, z) on the surface of the ellipsoid must satisfy the following equation [CR11, p. 17]:

$$\frac{x^2}{r_x^2} + \frac{y^2}{r_y^2} + \frac{z^2}{r_z^2} = 1.$$

The choice of $\mathbf{r}_{ellipsoid}$ is an important aspect for rendering globes. The *WGS84 ellipsoid* is defined with the radii $(6378137, 6378137, 6356752.314245)$ and is commonly used in other virtual globe systems [CR11, p. 19]. As will be explained later in chapter "StreamingATLOD", we use smaller radii for StreamingATLOD in order to avoid precision issues during rendering.

2.2.2 The WGS84 Coordinate System

The *World Geodetic System 84 (WGS84)* coordinate system (EPSG:4326) is a geographic reference system that was introduced in 1984.

A geodetic coordinate in WGS84 consists of three components:

- The *longitude* is measured in degrees in a range from $-180°$ to $180°$ and goes from west to east. The longitude at $0°$ is called the *Prime Meridian* and lies roughly where Greenwich, UK is.

- The *latitude* is measured in degrees as well in a range from $-90°$ to $90°$ and goes from south to north. The latitude at $0°$ is called the *Equator* and lies in the middle between the North and South Pole.

In order to render globes, geodetic coordinates must be transformed into cartesian coordinates and vice-versa. The various conversion formulas and their derivations can be found in chapter "Math Foundations" in the book *3D Engine Design for Globe Rendering* by Cozzi and Ring [CR11, p. 13].

2.2.3 The Web Mercator Projection

Even though the Earth is not flat, for some purposes it is easier to treat the Earth as a flat plane, for example when rendering 2D maps. A *map projection* projects geodetic coordinates from a spherical or ellipsoidal model onto a flat plane. The main map projection used in this thesis is the *Web Mercator projection*. It distorts the scale at the poles, making the North and South Poles famously appear larger than they really are, as shown in figure 2.4.

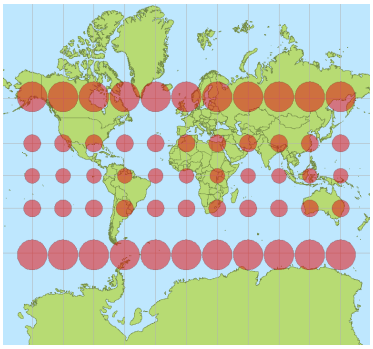


Figure 2.4: Tissot's indicatrix showcasing the distortion in the Mercator projection (image source: [Wik24c]).

The Web Mercator projection is based on the regular *Mercator projection* and was introduced and popularized by Google Maps. The main difference between the Mercator and the Web Mercator projection lies in the fact that the Web Mercator projection cuts off the latitude at $\pm 85.051129°$ in order to make images fit into a square [Wik24d].

The formula for the Web Mercator projection yields the projected (x, y) coordinates given a longitude and a latitude [con24]:

$$\begin{aligned} x &= lon, \\ y &= \ln\left(\tan\left(\frac{\pi}{4} + \frac{lat}{2}\right)\right). \end{aligned}$$

The *Inverse Web Mercator projection* yields the longitude and latitude given projected (x, y) coordinates [con24]:

$$\begin{aligned} lon &= x, \\ lat &= \frac{\pi}{2} - 2\arctan(\exp(-y)). \end{aligned}$$

2.2.4 Map Tiling

Managing and rendering maps and geographic data requires that they are split up into manageable segments with multiple resolutions. This process is called *map tiling* and is essentially the concept of LOD applied to geographic data. It is usually applied to images, such as cartographic or satellite images, or chunks of geospatial data inside a bounding rectangle. A *tile* refers to a single image or chunk at a particular position and a *zoom level*. The term *zoom level* is equivalent to the term *LOD level* in LOD rendering.

There exist numerous standards for map tiling, such as *XYZ tiles* (also called *ZXY tiles*) by Google Maps, the *Tile Map Service (TMS)* by OpenLayers, the *Web Map Tile Service (WMTS)* by the Open Geospatial Consortium (OGC), *Quadkeys* by Bing Maps, and more. All those standards organize the tiles hierarchically, such that each tile has four child tiles. Figure 2.5 shows a comparison of tile indexing methods for different tiling standards.

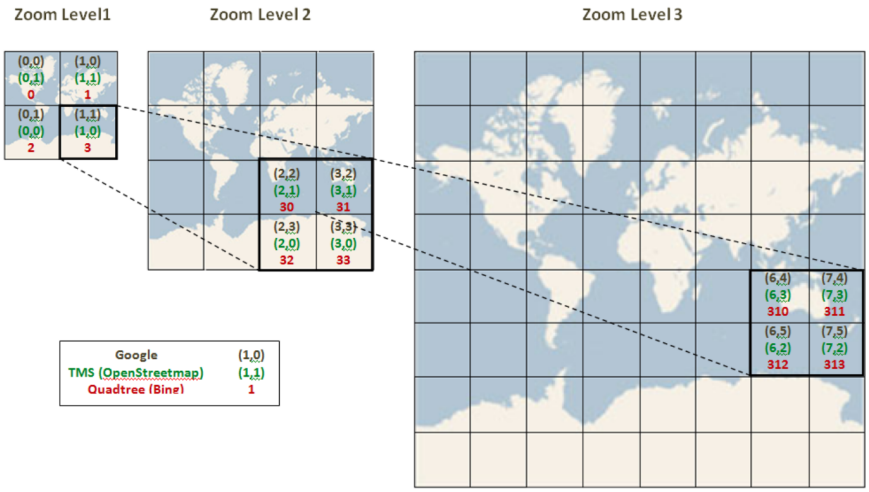


Figure 2.5: A comparison of tile indexing methods for Google Maps, Bing Maps and TMS (image source: [Pet12]).

For this thesis, the most relevant map tiling scheme is the *XYZ tiling scheme*. Each of the four child tiles represents a fourth of the parent tile's area, but at twice the resolution. Usually, every tile has the same pixel resolution, regardless of the current zoom level. Each tile has an x, y, and z-coordinate associated with it. The full triple is called the *tile key* and is denoted $tk = (tk_x, tk_y, tk_z)$ in this report. The x and y coordinates represent the position of the tile horizontally and vertically respectively, and the z-coordinate denotes the zoom level of the tile. Each tile can be uniquely identified with a single tile key. The (x, y) coordinates begin from the top-left corner and go to the bottom-right corner and the z coordinate starts from 0 (lowest resolution) and grows up to a maximum zoom level. The four child tiles of a given tile (tk_x, tk_y, tk_z) are computed as follows:

- Top-left child tile: $(2 \times tk_x, 2 \times tk_y, tk_z + 1)$

- Top-right child tile: $(2 \times tk_x + 1, 2 \times tk_y, tk_z + 1)$
- Bottom-left child tile: $(2 \times tk_x, 2 \times tk_y + 1, tk_z + 1)$
- Bottom-right child tile: $(2 \times tk_x + 1, 2 \times tk_y + 1, tk_z + 1)$

As a basic example, consider the scenario of having XYZ tiles of satellite imagery of the Earth. The zoom level 0 contains a single tile, the root tile, which has the XYZ tile key of $(0, 0, 0)$ and contains the entire Earth at the lowest resolution. The four child tiles at zoom level 1 are the top-left tile $(0, 0, 1)$, the top-right tile $(1, 0, 1)$, the bottom-left tile $(0, 1, 1)$ and the bottom-right tile $(1, 1, 1)$.

XYZ tiles are ususally served on web servers with an URL scheme of `http://www.example.com/z/x/y.ext`, where `z` corresponds to the zoom level, `x` and `y` to the tile coordinates and `.ext` to the file extension, such as `.png` or `.jpg` [Wik24b].

Chapter 3

Previous Work

This chapter aims to outline some already existing approaches and work. First, an overview of algorithms, literature and software systems for streaming-based terrain rendering is given, in which some of their central ideas are outlined. Afterwards, the author's predecessor project on basic terrain LOD rendering is summarized.

3.1 Terrain LOD Algorithms

In this section, a few terrain LOD algorithms are introduced. The reasoning behind choosing these algorithms is that StreamingATLOD relies on a few techniques introduced by these algorithms. Older terrain LOD algorithms which rely on continuous LOD, such as ROAM [DWS+97] or SOAR [LP01] are left out, since they are not relevant for this thesis.

3.1.1 Geometrical MipMapping

Geometrical MipMapping (also called *GeoMipMapping*) is a discrete terrain LOD algorithm introduced by de Boer [dB00] in 2000. The idea is to split up the terrain into square blocks of side length $2^n + 1$ for some $n \in \mathbb{N}$ and to create multiple meshes for different LOD resolutions of the same block. Each block has an AABB for view-frustum culling and a maximum geometrical error for the LOD level computation.

A quadtree is used for accelerating view-frustum culling in which the blocks are stored at the leaf nodes. Cracks between adjacent blocks with different LOD levels are prevented by ommitting the crack-causing vertices from the higher resolution mesh.

GeoMipMapping can be implemented with streaming by traversing all terrain blocks on the disk and generating a quadtree that stores the AABB of all blocks in the leaf nodes. The blocks required for rendering are then loaded using the AABBs stored in said quadtree.

The main advantage of GeoMipMapping is that it is simple to understand and

implement. The main disadvantage is that it scales poorly, since each block has the same size, making extremely large terrains more difficult to implement without special modifications.

3.1.2 Chunked LOD

Chunked LOD is a hierarchical terrain LOD algorithm introduced by Ulrich in the year 2002 [Ulr02]. The idea of Chunked LOD is to organize the terrain into a quadtree of chunks. Each chunk represents a section of the terrain at a particular LOD level, where the LOD level is given by the chunk's depth in the quadtree. Each non-leaf chunk has four children, where each child chunk represents a fourth of its parent's area, but in twice the resolution. The root chunk represents the entire terrain at the lowest resolution.

The heightmap and textures are preprocessed and organized into their respective chunks. During preprocessing, the section of the heightmap inside a chunk's bounds is compressed into a TIN.

Cracks between adjacent chunks with different LOD levels are hidden by rendering skirts around each chunk. Popping artifacts are prevented through geomorphing. Chunked LOD supports streaming and is designed to render the already available data in case the requested data is not loaded yet.

The main advantage of Chunked LOD is that it scales well and allows for massive terrains to be rendered thanks to its hierarchical nature. The main disadvantage is that it requires some lengthy preprocessing of the terrain data.

3.1.3 Geometry Clipmaps

Geometry Clipmaps is a discrete terrain LOD algorithm introduced by Hoppe and Losasso in 2003. A follow-up GPU-based variant of Geometry Clipmaps (called *GPU-based Geometry Clipmaps*) was published by Hoppe and Asirvatham [AH05] in the year 2004. In this subsection, GPU-based Geometry Clipmaps will be introduced.

The idea of GPU-based Geometry Clipmaps is to render a ring of constant grid-like flat meshes centered around the camera and to displace the height values of the meshes in the vertex shader using a heightmap that was loaded in advance as a texture on the GPU, a concept which has since been dubbed *heightmap displacement*. The scale of the meshes increases the further away from the camera they are, resulting in fewer vertices for distant areas.

Cracks are avoided by rendering degenerate triangles and popping artifacts are avoided through geomorphing in the vertex shader. At close distances, the terrain is augmented with procedural detail generated in the vertex shader.

Hoppe and Asirvatham did not discuss streaming of terrain data. Instead, they mainly relied on texture compression, with which they compressed a 20-billion sample heightmap of the United States into 355 MB and decompressed required sections at runtime for rendering.

The main advantage of GPU-based Geometry Clipmaps is that very little memory is used for the meshes thanks to heightmap displacement. The meshes only

need to be defined once at start up and stay constant throughout the runtime. The main disadvantage is that GPU-based Geometry Clipmaps does not incorporate height for LOD, since the meshes are always centered around the camera.

3.2 Reference Books

3.2.1 3D Engine Design for Virtual Globes

The book *3D Engine Design for Virtual Globes* by Cozzi and Ring [CR11] is a reference book for various topics related to globe rendering. It includes chapters on mathematical basics of geographic coordinate systems, terrain rendering basics, handling multithreaded rendering, solving precision issues and more.

3.2.2 Level of Detail for 3D Graphics

Subchapter "7.2.5 Paging, Streaming, and Out of Core" in the the book *Level of Detail for 3D Graphics* [LWC+02] by Luebke et al. describes some basic background, existing approaches and relevant publications related to terrain rendering and streaming.

3.3 Video Games and Game Engines

3.3.1 Microsoft Flight Simulator

Microsoft Flight Simulator is a video game developed by Asobo Studio and released in 2020. The video game allows the player to fly on the entire earth and uses various data sources for representing the earth as accurately as possible. It received high praise for its realism and immersion.

At the GDC 2022, Fuentes presented the terrain system of Microsoft Flight Simulator, in which he described the data organization, system architecture, rendering process and many more aspects [Fue22]. The terrain system utilizes a hierarchical LOD approach based on quadtrees and renders skirts to hide cracks, similarly to Chunked LOD. It uses a large number of sources for height data, TINs, overlay texturing, vegetation, bodies of water, cities and more. The data and its states are managed with a complex state-machine-based marker system, where each piece of data of a tile (such as height data, TIN, etc.) is a state machine. It also utilizes Microsoft Azure cloud for computing road data, tree masks and other features.

3.4 Globe Renderers

3.4.1 Google Earth

Google Earth is a 3D globe viewer developed by Google released in the year 2001 and is most likely the most well-known example of a globe renderer. It is capable of rendering the Earth by utilizing a number of data sources for its imagery and height data. Google Earth supports adjustable cache sizes for the disk cache and the memory cache.

3.4.2 OpenWebGlobe

OpenWebGlobe [LCN12] is a software development kit (SDK) developed by the Institute for Geomatics at the University of Applied Sciences of Northwestern Switzerland. It is written in JavaScript and WebGL and designed for rendering 3D globes on web browsers. The source code of OpenWebGlobe is open source.

OpenWebGlobe's terrain data is preprocessed into TINs using a large-scale Delaunay triangulation using raw height data and subsequently organized into XYZ tiles. The triangulation was performed on a large-scale cluster of computers.

The Earth is rendered using a Chunked LOD-based hierarchical LOD algorithm using the preprocessed TINs and overlay textures served from web servers. Multiple height and overlay layers are supported.

3.4.3 Cesium

Cesium Geospatial is a company developing globe-rendering-related software. Their open-source JavaScript and WebGL SDK *Cesium JS* also uses a hierarchical LOD algorithm. For terrain data representation, they developed the *quantized-mesh* file format specifically designed for streaming.

3.5 Review of Project 2 – 3D Terrain with Level of Detail

This thesis is the logical continuation of the author's preceeding project "3D Terrain with Level of Detail" [Tab24] as part of the project course "Project 2" at the Bern University of Applied Sciences. In said project, the basics of terrain rendering and existing approaches for terrain LOD were studied and a simple terrain renderer named *ATLOD* was developed. Terrain rendering with streaming was explicitly left out of scope of the preceeding project.

3.5.1 Summary of the Implemented Terrain LOD Algorithm

The implemented LOD algorithm is mainly based on GeoMipMapping [dB00], but performs the rendering with heightmap-displacement in the vertex shader, as introduced by GPU-based Geometry Clipmaps [AH05]. The terrain gets split up into blocks of size $blockSize = 2^n + 1$ for some $n \in \mathbb{N}$. Each block contains information related to a particular section of terrain, such as current LOD level, its world-space center point, its AABB, current border permutation. A single grid-like flat mesh of side length $blockSize$ is stored on a single global vertex and index buffer. This flat mesh will be used to render every block at runtime. The indices are stored such that the flat mesh is split into its center and border area. The border area is defined in a special manner in order to prevent cracks from occuring.

The first part of the index buffer contains indices of the center area at every LOD resolution, from highest to lowest, and the second part of the index buffer contains the indices of the border area at every LOD resolution and for every

of the $2^4 = 16$ possible border permutations. A border permutation is a 4-tuple (l, r, t, b) (the variables corresponding to left, right, top and bottom) representing a border area, set up such that each element of the 4-tuple is set to 1 if the neighboring block on the corresponding side has a lower LOD level, otherwise 0. The maximum difference in LOD level between any two neighboring blocks must be 1. The heightmap is stored as a texture object on the GPU.

At runtime, in a first step the LOD level and current border permutation of each block is updated. The new LOD level is calculated using the distance between the camera and the block's center point. In a second step, each block is intersected with the view-frustum and if the block's AABB is inside the view-frustum, the block gets rendered with two draw-calls, one for the center area and one for the border area.

In the vertex shader, each vertex of the flat mesh gets translated by the block's world-space center point and the heightmap is sampled for a height value with the (x, z) coordinates of the vertex, which is then used to displace the y-coordinate. In the fragment shader, the overlay texture is applied and Phong shading is calculated. The normal vectors for Phong shading are calculated using the four orthogonally neighboring points in the heightmap. Finally, a simple distance fog is applied.

3.5.2 Results

The renderer worked well on the tested hardware (Apple Intel MacBook Air 2020), consistently rendering around 60 frames per second with a 14000×14000 terrain. A screenshot of a rendered terrain is shown in figure 3.1.

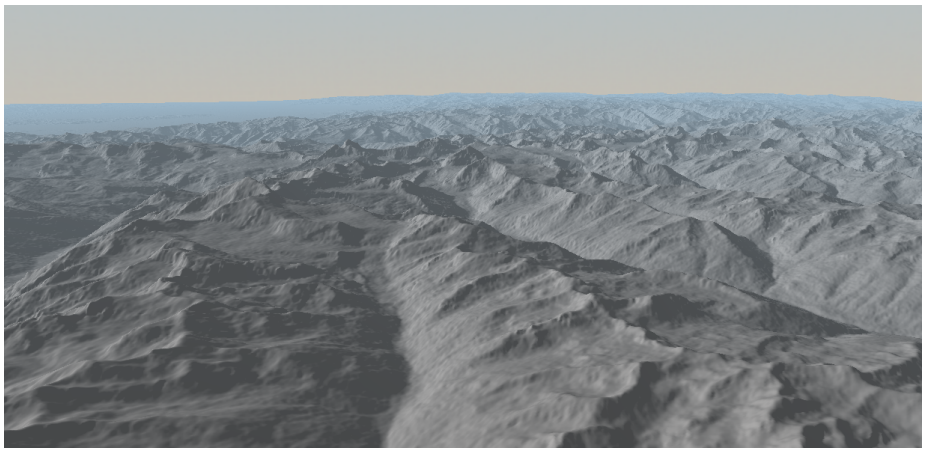


Figure 3.1: Screenshot of a terrain rendered in ATLOD.

The main strengths of the implemented terrain LOD algorithm are its low GPU memory usage, since each block uses the same mesh, and its simplicity to understand.

Some improvement points in terms of visual and rendering performance were to render the terrain with geomorphing to avoid popping artifacts, *instanced*

rendering for reducing the number of draw calls and improving view-frustum culling by organizing AABBs in a quadtree.

The most important missing technical feature, however, was support for streaming, which would allow for even larger terrains to be rendered.

Chapter 4

StreamingATLOD: Streaming-Assisted Terrain Level of Detail

This chapter describes the implementation of *StreamingATLOD*: **Streaming-A**ssisted **T**errain **L**evel **o**f **D**etail. First, some preliminary information, such as the used programming languages, tools, libraries, and data will be given. Afterwards, a high-level overview of StreamingATLOD will be presented. The missing details from the high-level overview will be filled section by section, starting from the main classes and data structures, followed by geographic conversions, the used meshes, the LOD metric and the culling techniques. Subsequently, the streaming and caching mechanisms will be presented and rounded off with a deep dive into the rendering process, thereby finally filling the last piece of missing detail. At the end, a few miscellanous features are mentioned, such as the camera, the collision detection and the configuration file support.

4.1 Preliminaries

4.1.1 Technologies

StreamingATLOD is written in C++17 and OpenGL 4.0. For compiling build files, CMake (minimum version 3.5) is used. The source code of StreamingATLOD is hosted on GitHub under the repository `AmarTabakovic/bachelor-thesis`. StreamingATLOD uses the following third-party libraries:

- GLM: The *OpenGL Mathematics (GLM)* library provides types, functions and constants for the mathematics of computer graphics. It is designed to syntactically ressemble the mathematical features of GLSL, such as vectors, matrices and transforms.
- GLEW: The *OpenGL Extension Wrangler* library is an extension loading library for OpenGL.

- GLFW: *GLFW* is a multi-platform library for desktop-based OpenGL applications, offering an operating-systems-agnostic API for managing windows, contexts and input handling.
- Dear ImGui: *Dear ImGui* is a multi-platform graphical user interface library. StreamingATLOD uses Dear ImGui for a minimalistic user interface.
- STB: *STB* is a collection of header-only libraries developed by Sean Barrett. StreamingATLOD uses the header `stb_image.h` for loading images.
- libwebp: *WebP* is an image format developed by Google designed for images on the web. Its C++ API *libwebp* is used for loading and decoding WebP images.
- libcurl: *cURL* is a command-line utility for issuing client-side HTTP-requests. StreamingATLOD uses its C++ API *libcurl* mainly for downloading the terrain data from web APIs.

4.1.2 Terrain Data Provider

The satellite imagery and elevation data is provided by web-based Cloud APIs by the company *MapTiler*. The datasets "Satellite V2" and "Terrain RGB V2" are used, both of which are organized with the XYZ tiling scheme and projected to Web Mercator. The "Satellite V2" dataset consists of satellite imagery up to zoom level 22 and is served in JPEG format. The "Terrain RGB V2" dataset consists of elevation data up to zoom level 14 and is served in WebP format. The satellite data will be used up to level 14, so that there is a 1 to 1 correspondence of satellite tiles and elevation tiles. Zoom level 14 corresponds to a precision of 9.5 meters per pixel [Map]. This is important for the implemented terrain LOD algorithm, which will be described shortly.

For this thesis, the "Flex" plan of MapTiler Cloud was subscribed to, which costs $25 per month. The plan includes 500'000 API requests per month and $0.10 per 1000 extra requests when exceeding the monthly 500'000 requests.

The Terrain RGB dataset encodes height values with RGB using a special encoding and decoding formula [Map23]. The following formula converts decodes an RGB triple with each color channel containing values in the range $0, \ldots, 255$ into a height value in meters above sea level:

$$height = -10000 + ((r \cdot 256^2 + g \cdot 256 + b) \cdot 0.1).$$

4.1.3 Basic Renderer Design

The basic renderer design, such as the `Shader` class and the structure of the `Camera` class, is based mainly on *Learn OpenGL* by de Vries [dV20]. The `Shader` class represents a single OpenGL shader program with a vertex shader and a fragment shader. It has a field `unsigned _id` and various methods to set uniform variagles.

The `Camera` class represents the movable camera and is modified slightly in order to allow for better traversal of the Earth. It will be described in greater detail in section "Miscellaneous Features".

4.2 System Overview

This section aims to first outline the terrain LOD algorithm used for StreamingATLOD and subsequently give a high-level overview of the system architecture. The concrete implementation details are described in greater detail in later sections, slowly filling in the gaps left by the high-level overview.

4.2.1 Terrain LOD Algorithm

The implemented terrain LOD algorithm is mainly based on *Chunked LOD*[Ulr02]. Each quadtree node (from now on callled *terrain node* or simply *node*) has either four children or no children. Each child node represents a fourth of the parents area but at twice the resolution. The main difference to the original Chunked LOD algorithm is that StreamingATLOD does not preprocess the heightmap and compute a TIN per quadtree node. Instead, we store a heightmap on each node, upload it to the GPU as a texture object and use heightmap-displacement in the vertex shader, as described in greater detail in sections "Meshes" and "Rendering".

One major advantage of Chunked LOD is that it maps very straightforwardly to map tiling schemes (in particullar the XYZ tiling scheme), which are also organized hierarchically. Each terrain node can be uniquely mapped to a satellite overlay tile and a heightmap tile. Conversely, each satellite tile and heightmap tile can be uniquely mapped to a terrain node. Additionally, each terrain node can be uniquely identified by its XYZ tile key, which allows for constant-time random access using associative data structures such as hashmaps or hashsets, which will be useful later. Figure 4.1 illustrates this relation between terrain nodes and tiles.

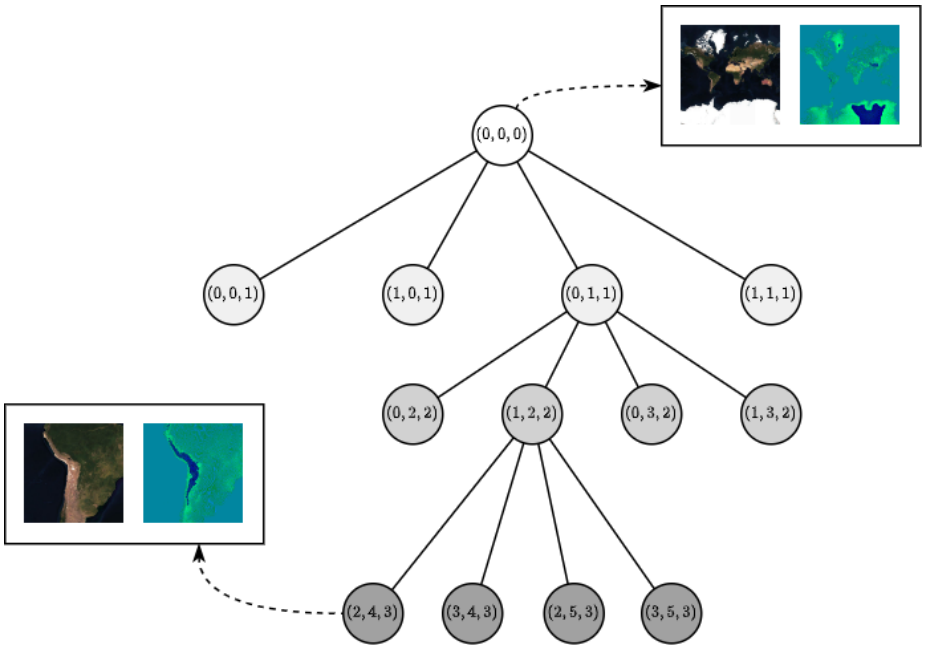


Figure 4.1: A terrain node tree and the associated terrain data (overlay texture and heightmap) that each terrain node stores.

At runtime, two main steps happen:

1. Collection traversal: The first step is the collection of the list of all visible nodes for rendering. For this, the terrain nodes get traversed recursively starting from the root node. If a traversed terrain node is visible and the traversal should not continue with the node's children, the node gets added to the render list. If a node's children are not loaded in memory, they get requested and the current node is added to the render list as a replacement.
2. Rendering: The second step is the actual rendering of the collected visible nodes. For this, the list of visible nodes gets iterated and for each node, the textures get bound, the uniforms get updated and the necessary meshes get rendered.

The details of the rendering process will be described in greater detail in section "Rendering Process".

4.2.2 System Architecture

StreamingATLOD is built with a multithreaded architecture based on worker threads and message queues for inter-thread-communication. Figure 4.2 shows a high-level overview of the system architecture. The circled numbers are described in the numbered list further below.

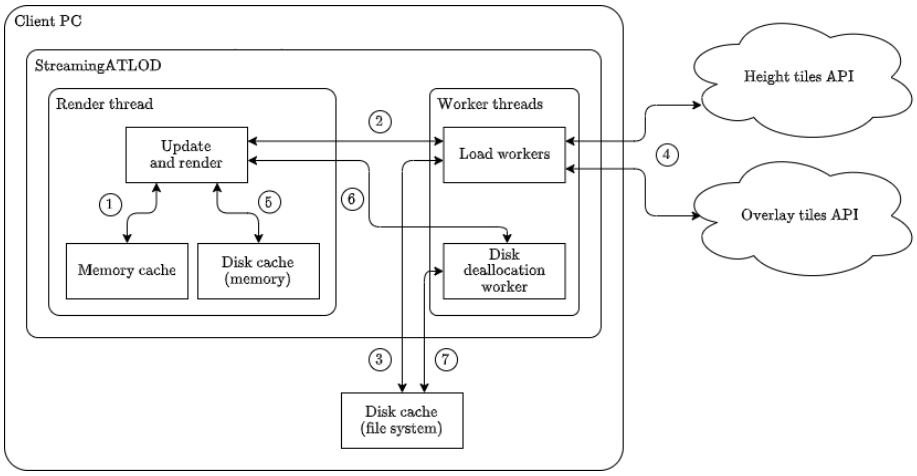


Figure 4.2: High-level overview of StreamingATLOD's system architecture.

The *render thread* is the main thread where the updating and rendering of the terrain happens. It issues all OpenGL commands, including loading various vertex and index buffers at startup, generating new textures on the GPU, updating uniform variables, performing draw calls and deallocating unused textures from the GPU. The render thread also manages the *memory cache* and the *disk cache*. The memory cache stores all terrain nodes which are currently loaded in memory and ready to be rendered. The disk cache consists of the *file system disk cache*, which is a folder on the file system that contains the actual heightmaps and overlay images, and the *in-memory disk cache*, which tracks the tile keys of the data that is stored in the file system disk cache. Both the memory cache and the disk cache are *least-recently used (LRU)* caches with a fixed capacity, which means that the least-recently used entries are evicted upon exceeding the capacity. The disk cache has a much larger capacity than the memory cache. The LRU cache implementation will be described in greater detail in section "Main Classes Data Structures".

The *worker threads* are separate threads responsible for tasks that might stall the render thread, such as disk or network I/O. The main thread and the worker threads communicate with eachother through *message passing*. Each worker thread has its own *request queue* for receiving requests, which the main thread can use to send requests to the worker threads. Each worker thread also stores a reference to the main thread's *done queue* for sending back responses. Since both the main thread and the worker threads access the queues, the message queues are *concurrent message queues* which lock during each operation for thread-safety. Like the LRU cache implementation, the concurrent message queue implementation will be described in greater detail in section "Main Classes and Data Structures" as well.

There are two types of worker threads employed by StreamingATLOD: the *load workers*, which load in terrain data from the disk cache or from the web APIs, and the single *disk deallocation worker*, which deletes unneeded data from the disk cache. The terrain nodes in the memory cache and disk cache must be handled carefully in order to prevent race conditions and inconsistent

application states, such as having a terrain node both in the load worker and the disk deallocation worker at the same time. The implementation details of the worker threads, including the careful handling of terrain nodes, is described in greater detail in section "Terrain Streaming".

Note that neither worker thread type performs any OpenGL operations, since OpenGL calls must happen in the same thread where the context is defined [Inc]. Instead, all OpenGL operations are performed in the render thread as previously mentioned.

The way that the components and threads work with each other is described in the numbered list below. The numbers are based on figure 4.2:

① Each render frame, the visible terrain nodes are traversed and collected for rendering. The memory cache stores the most recently used terrain nodes that are ready to be rendered. During the collection traversal, we check whether the memory cache contains the desired terrain node, and if so, add it to the list of nodes to be rendered. Otherwise, we proceed with ②. Additionally, each render frame we process newly loaded terrain nodes that come off the load worker done queue and store them in the memory cache.

② If during the collection traversal in ① the memory cache did not contain a desired node, the main thread sends a request to a load worker by putting the request on the load worker's request queue. Putting a request into the request queue is a non-blocking operation, which means the render thread can continue the collection traversal and rendering. If there are multiple load workers operating, an integer is used to track the load worker which was the most recently used and incremented every time a new request is performed. As a result, requests are cyclically distributed over the load workers in a round-robin fashion. The load worker request contains the XYZ tile key of the desired node and whether to load the data from the disk cache or from the web API. The load worker then processes the request either by retrieving it the heightmap and imagery of the desired node from the disk cache ③ or from the tile web API ④. The load worker also creates the new terrain node object and generates the AABB. Finally, the load worker sends back a response. The response can be successful, containing the terrain node instance and, the heightmap and the overlay imagery ready to be loaded and rendered. It can also send back a failed response, such as when there is a network error or a request timeout, or a so-called *unloadable* response, for when the requested terrain node does not exist (e.g. oceans at high zoom levels).

③ In ② the load worker received a request for loading a terrain node. Based on the request type, the load worker either loads the data from the disk cache on the file system or from the web API. In the current numeration, the load worker interacts with the file system cache in two ways. The first way is by storing new terrain data that was requested from the web API in the file system disk cache. The second way is by reading and loading terrain data from the file system disk cache into memory after the main thread requested them from the disk cache.

④ The load worker communicates with the web APIs after the main thread

sent out a request for terrain data from the web APIs. For this, the load worker performs two HTTP requests, one for the heightmap and one for the overlay texture. If at least one of both failed, whether due to an error or due to the node being unloadable, the load worker sends back the appropriate response so that the main thread can handle this scenario. Otherwise, the load worker continues constructing the response as described in ②.

⑤ Every new terrain node that was downloaded from the API gets put into the in-memory disk cache after being processed from the done queue. Additionally, during the collection traversal, the in-memory disk cache gets accessed in order to check whether a terrain node is available on the disk cache.

⑥ Throughout the lifetime of StreamingATLOD, the disk cache grows. Once the maximum capacity has been reached, the least-recently used entries must be evicted from the disk cache. This includes deleting the entry from the file system disk cache. For this, the render thread sends a request with an XYZ tile key to the disk deallocation worker, after which the disk deallocation worker deletes the data from the file system in ⑦. Afterwards, the disk deallocation worker sends a response back to the render thread to signal that it has deleted the data.

⑦ The disk deallocation worker deletes the actual terrain data on the disk.

4.3 Main Classes and Data Structures

This section describes some of the central classes and data structures used in StreamingATLOD, starting with the essential XYZ tile key and terrain node classes, followed by the terrain manager class where the majority of runtime operations happen. Afterwards, the implementation of the two key data structures LRU cache and concurrent message queue are described.

4.3.1 XYZ Tile Key Class

The XYZ tile key is represented by the class `XYZTileKey` and is of central importance, since it is used to identify each terrain node uniquely and is used as the key type for unordered data structures, such as the `std::unordered_map` or `std::unordered_set`. It has three `unsigned` fields `_x`, `_y` and `_z`. It also contains the four methods `topLeftChild()`, `topRightChild()`, `bottomLeftChild()` and `bottomRightChild()` to generate each of the four child tile keys. The `XYZTileKey` class implements a hash function and overrides the default comparison function in order to work with unordered data structures.

4.3.2 Terrain Node Class

A single terrain node is represented by the class `TerrainNode`. The terrain node class serves mainly as a simple container and does not include many methods for runtime behaviour. Its most important members include the following:

- `XYZTileKey _tileKey`: The XYZ tile key of the tile.

- `unsigned _overlayTextureId`: The texture ID of the overlay texture.
- `unsigned _heightmapTextureId`: The texture ID of the heightmap texture.
- `glm::vec3 _aabbP1, _aabbP2`: The two points representing the AABB.
- `std::vector<glm::vec3> _projectedGridPoints`: The vector containing nine ellipsoid-projected points on the tile arranged in a grid-like manner: the top-left, top-center, top-right, middle-left, center, middle-right, bottom-left, bottom-center, bottom-right points. These points are used for the LOD metric calculation described in greater detail later.
- `std::vector<glm::vec3> _horizonCullingPoints`: A vector similar to `_projectedGridPoints`, except that the vertices are projected to the ellipsoid using a slightly higher height value. These points are used for horizon culling as described in subsection "Culling".
- `unsigned char* _heightData`: The raw height data. This is used for collision detection.
- `std::chrono::system_clock::time_point _lastUsedTimeStamp`: The timestamp indicating when the tile was last rendered.

4.3.3 Terrain Manager Class

The class `TerrainManager` is the main workhorse of StreamingATLOD. All major runtime operations, such as updating, rendering, cache management, collision detection and more are performed in methods defined in `TerrainManager`.

The `TerrainManager` class has numerous members which can be categorized as follows:

- Meshes and shaders: The meshes and shaders of the terrain, such as the terrain mesh, skirt mesh and pole mesh, are members of `TerrainManager`, since `TerrainManager` is responsible for rendering them. The mesh types will be introduced in section "Meshes".
- Caches: `TerrainManager` holds the memory cache and the in-memory disk cache and also performs the deallocation for evicted for them.
- Workers and concurrent message queues: `TerrainManager` holds the worker thread instances and concurrent message queue instances for the load workers and the disk deallocation worker. It also holds the current load worker ID for the round-robin-style load worker scheduling.
- Various XYZ tile key lists: The tile keys of terrain nodes that are currently being loaded or that are currently being evicted from the disk are stored on `std::unordered_set<XYZTileKey>` instances, so that `TerrainManager` can quickly perform some important checks, e.g. to avoid accidentally requesting a terrain node that is already being loaded or to avoid requesting a terrain node that is currently being evicted from the disk cache.

The methods and usage of the member variables will be described in greater detail in the subsequent sections wherever they are used.

4.3.4 LRU Cache

The *least-recently used (LRU)* policy is a cache management policy which states that only the most-recently used data should be kept. It is one of many cache management policies used in various areas of computer science, such as operating systems, distributed systems and web technologies [Red].

The *LRU cache* is an associative data structure with a fixed capacity that operates according to the LRU policy. Inserting a new key-value pair, updating an existing key-value pair and retrieving an existing key-value pair results in the key-value pair being the most-recently used and thus the least-likely candidate for eviction. If the capacity has been reached and a new key-value pair is inserted, the least-recently used key-value pair gets evicted.

The LRU cache in StreamingATLOD is implemented as a simple template class `LRUCache<K,V>`, where `K` is the key type and `V` is the value type. Its capacity can be set in the constructor. The LRU cache uses a `std::unordered_map` and a `std::list` in the background, which is a common strategy for implementing LRU caches [Red]. It supports the following operations:

- `PutResult<K,V> put(const K& key, const V& value)`: This method adds the given key-value pair to the cache. If the cache is full, the least-recently used element is evicted and returned as a value of the type `PutResult<K,V>`. The struct `PutResult<K,V>` stores a `std::optional<std::pair<K,V>>`, which either contains the least-recently used key-value pair that was just evicted, or `std::nullopt` if the cache is not full yet. Figure 4.3 illustrates an example of an eviction after putting a new element into a full cache.

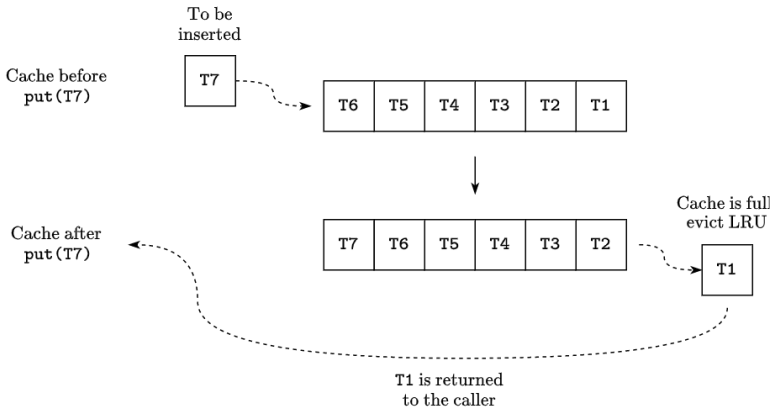


Figure 4.3: Insertion of the element `T7` with the eviction of the least-recently used element `T1`.

- `std::optional<V> get(const K& key)`: This method gets the value of

the given key. If the value doesn't exist, it returns `std::nullopt`. Additionally, the cache entry gets pushed to the front of the cache since it was just used. Figure 4.4 illustrates this example.

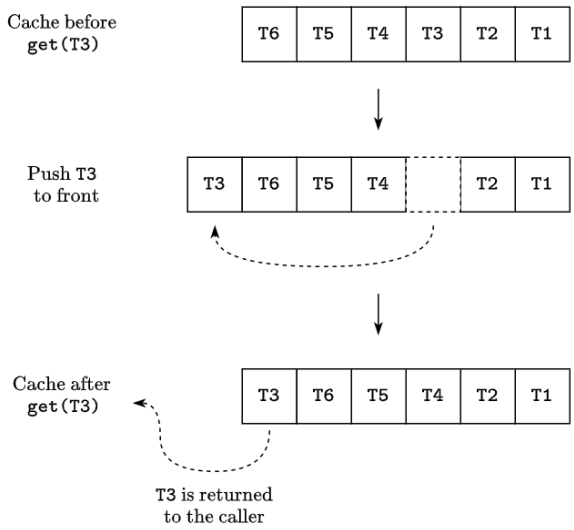


Figure 4.4: Retrieval of the element `T3`. Note that after the retrieval, `T3` is the most-recently used element and thus in the front of the cache.

- `bool contains(const K& key)`: This method checks whether an element with a given key exists in the LRU cache without affecting its eviction priority.

4.3.5 Concurrent Message Queue

The *concurrent message queue* is a thread-safe data structure designed for communicating between different threads. A thread can safely push messages to the queue, which can then be processed by another thread. The access to the queue is locked, such that only a single thread can access it at a time. Figure 4.5 shows an illustration of the concept of using worker threads.

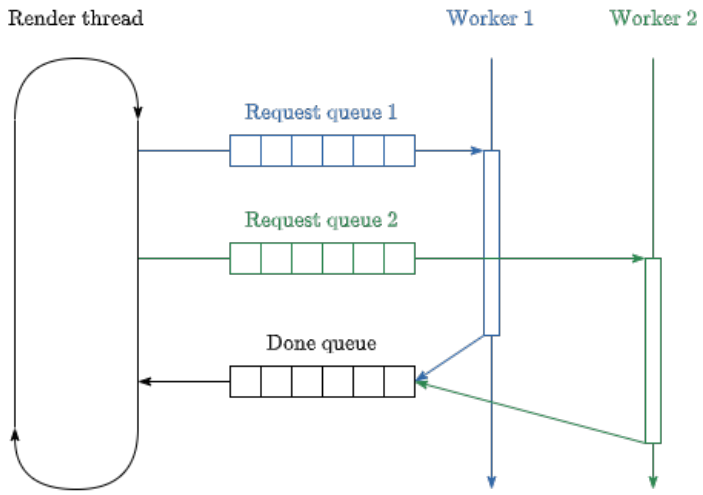


Figure 4.5: Illustration of the idea of worker threads with message queues. The main thread sends out requests to the request queue, which get processed by the workers in the background and then sent back to the render thread. The image is based on [CR11, p. 287]

StreamingATLOD's implementation of the concurrent message queue is loosely based on chapter "Exploiting Parallelism in Resource Preparation" in the book "3D Engine Design for Virtual Globes" by Ring and Cozzi [CR11, p. 275].

The concurrent message queue is implemented as a small template class `MessageQueue<T>`. It supports the following operations, which are all locked using a `std::mutex` for thread-safety:

- `void push(T message)`: Pushes a message to the concurrent message queue.
- `void pushAll(std::deque<T> messages)`: Pushes the contents of a given queue to the concurrent message queue.
- `std::optional<T> pop()`: Pops and returns the message at the front of the queue. If the queue is empty, `std::nullopt` is returned.
- `std::deque<T> popAll()`: Pops all messages and returns them in a `std::deque`.

The methods postfixed with `-All()` allow for quick processing of all elements in the queue by a thread without having to repeatedly lock the queue for each element.

4.4 Geographic Conversions

This section describes the implementation of the main geographic projections and conversions used in StreamingATLOD. All functions using longitudes and latitudes expect them in radians as input and return them as radians.

The map projections performed on the CPU are encapuslated in the C++ header

file `mapprojections.h`, which defines a namespace `MapProjections` with functions inside for performing map projections.

4.4.1 Web Mercator and Inverse Web Mercator

The Web Mercator and Inverse Web Mercator projections are encapsulated in the functions `webMercator()` and `inverseWebMercator()` respectively. Listing 4.1 shows the pseudocode for both the Web Mercator and the Inverse Web Mercator projections.

```
1  inline glm::vec2 webMercator(glm::vec2 lonLat)
2  {
3      float x = (glm::degrees(lonLat.x) + 180.0f) / 360.0f;
4      float y = 0.5f - glm::log(glm::tan(glm::pi<float>() / 4.0f + lonLat.y /
       2.0f)) / (2.0f * glm::pi<float>());
5
6      return glm::vec2(x, y);
7  }
8
9  inline glm::vec2 inverseWebMercator(glm::vec2 webMercXY)
10 {
11     float lon = glm::radians((webMercXY.x * 360.0f - 180.0f) * -1.0f);
12     float lat = glm::radians((glm::atan(glm::exp(glm::pi<float>() * (1.0f -
       2.0f * webMercXY.y))) * 2.0f - glm::pi<float>() / 2.0f) * 180.0f / glm::
       pi<float>());
13
14     return glm::vec2(lon, lat);
15 }
```

Listing 4.1: The functions `webMercator()` and `inverseWebMercator()` in `MapProjections`.

4.4.2 Geodetic-to-cartesian Transformations

As mentioned previously, the geodetic-to-cartesian transformations are based on chapter "Math Foundations" of the book "3D Engine Design for Virtual Globes" by Cozzi and Ring [CR11, p. 13]. The first two functions are `geodeticSurfaceNormal()`, which both compute the surface normal on an ellipsoid given a geodetic coordinate. The first function takes a 3D point and the squared ellipsoid radii, whereas the second point only takes the squared ellipsoid radii. The third function is `toGeodetic2D()`, which computes the 2D geodetic coordinate (lon, lat) given a 3D cartesian coordinate and the squared ellipsoid radii. The fourth function is `geodeticToCartesian()`, which given a 2D geodetic coordinate (lon, lat) and the squared ellipsoid radii computes the cartesian coordinate on the ellipsoid. All four functions are shown in listing 4.2.

```
1  inline glm::vec3 geodeticSurfaceNormal(glm::vec3 p, glm::vec3
       globeRadiiSquared)
2  {
3      glm::vec3 normal = p * (glm::vec3(1.0f) / globeRadiiSquared);
4      return glm::normalize(normal);
5  }
6
7  inline glm::vec3 geodeticSurfaceNormal(glm::vec3 geodetic)
8  {
9      float cosLat = glm::cos(geodetic.z);
```

```
10
11      return glm::vec3(cosLat * glm::cos(geodetic.x), glm::sin(geodetic.z),
        cosLat * glm::sin(geodetic.x));
12  }
13
14  inline glm::vec2 toGeodetic2D(glm::vec3 position, glm::vec3 globeRadiiSquared
        )
15  {
16      glm::vec3 n = geodeticSurfaceNormal(position, globeRadiiSquared);
17      return glm::vec2(glm::atan2(n.z, n.x) * -1, glm::asin(n.y / glm::length(n
        )));
18  }
19
20  inline glm::vec3 geodeticToCartesian(glm::vec3 globeRadiiSquared, glm::vec3
        geodetic)
21  {
22      glm::vec3 n = geodeticSurfaceNormal(geodetic);
23      glm::vec3 k = globeRadiiSquared * n;
24      float gamma = glm::sqrt(k.x * n.x + k.y * n.y + k.z * n.z);
25
26      glm::vec3 rSurface = k / gamma;
27      return rSurface + (geodetic.y * n);
28  }
```

Listing 4.2: The functions `geodeticSurfaceNormal()`, `toGeodetic2D()` and `geodeticToCartesian()` in `MapProjections`.

4.4.3 On the GPU

The above described functions were implemented in the vertex shaders as well where needed. The functions in the shaders are essentially the same, except that instead of in C++, they are written in GLSL. Listing the code for these functions again but in GLSL would be redundant.

4.4.4 Choice of Ellipsoid Radii

As mentioned in "Theoretical Background", StreamingATLOD does not use the WGS84 ellipsoid with radii $(6378137, 6378137, 6356752.314245)$ for geodetic-to-cartesian conversions. Instead, it uses an ellipsoid of much smaller scale of $\sqrt{(100000, 100000, 100000)} = (316.2277, 316.2277, 316.2277)$ in order to avoid precision issues. This value was determined entirely through experimentation, such that there was enough precision for vertex transforms and enough precision for the depth buffer. Since the terrain data is used up to zoom level 14 anyway, which corresponds to a maximum precision of around 9.5 meters[Map], specialized solutions for precision issues such as the ones described in chapter "Theoretical Background" are currently not needed. Incorporating measures for handling higher precision terrain data is a potential improvement point for the future.

4.5 Meshes

This section introduces the three main mesh types used for rendering the Earth: the *terrain mesh*, the *skirt mesh* and the *pole mesh*. Each mesh type is encapsulated in its own class (`GridMesh` for the terrain mesh, `SkirtMesh` for the skirt mesh and `PoleMesh` for the pole mesh).

4.5.1 Terrain Mesh

StreamingATLOD renders the terrain by displacing the height values of a flat mesh in the vertex shader, similarly to GPU-based Geometry Clipmaps [AH05] and the basic ATLOD[Tab24]. This flat mesh, from now on called the *terrain mesh* or *grid mesh*, is a flat square grid-like mesh of side length $n \in \mathbb{N}$ centered around $(0, 0, 0)$. Each vertex consists of a (x, y) coordinate and the coordinates go from $(-n, -n)$ to (n, n). The terrain mesh is organized with triangle strips, so that it can be rendered as the OpenGL `GL_TRIANGLE_STRIP` primitive. The main advantage of using triangle strips is that they use up less GPU memory in comparison to e.g. `GL_TRIANGLES`. Each row of the mesh is separated with a special marker index named `RESTART`, which is set to the maximum `GLuint` value. The reason for this is that when starting a new row, the indices of the row can be defined immediately after using `RESTART`, instead of the alternative approach of having to define so-called *degenerate triangles* in order to start a new row. Figure 4.6 shows an illustration of a 3×3 terrain mesh.

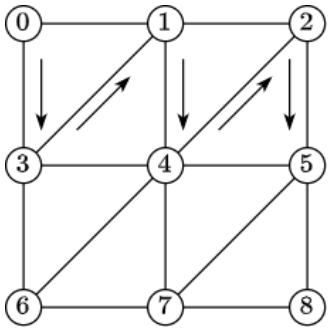


Figure 4.6: Illustration of a flat 3×3 terrain mesh viewed from the top. The indices are organized as triangle strips in the following order: 0, 3, 1, 4, 2, 5, `RESTART`, 3, 6, 4, 7, 5, 8, `RESTART`.

In the vertex shader, the terrain mesh first gets its heights displaced using the heightmap that was loaded on the GPU, after which it is projected to the ellipsoid with the Inverse Web Mercator projection and the geodetic-to-cartesian transformation. These operations are described in greater detail in the later section "Rendering".

The terrain mesh's side length n can be chosen arbitrarily, since textures can be filtered bilinearly on the GPU. This also means that the terrain mesh can be defined in multiple resolutions. StreamingATLOD supports three different terrain mesh side lengths (`low`, `medium` and `high`) which can be configured in a configuration file as described in the later section "Miscellaneous Features". The terrain mesh resolution `low` is currently rendered for zoom levels 0 to 8, `medium` for zoom levels 9 to 11 and `high` for zoom levels 12 and above.

4.5.2 Skirt Mesh

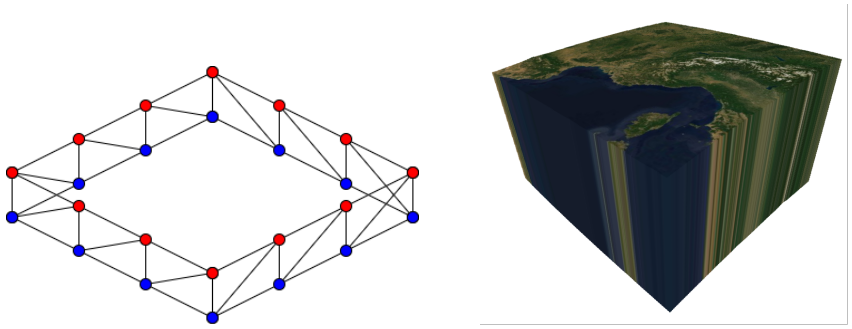
In order to avoid cracks from occurring StreamingATLOD uses skirts as introduced in chapter "Theoretical Background". For this, StreamingATLOD uses

a special *skirt mesh*.

The skirt mesh is defined almost exactly the same as the terrain mesh, having the same side length as the terrain mesh being also organized as triangle strips. The first main difference is that the skirt mesh only consists of the outermost vertices at the perimeter of the terrain mesh, whereas the terrain mesh consists of the entire filled $n \times n$ grid.

The second main difference is that the vertices are defined in a special way. A skirt mesh vertex consists of a (x, z) coordinate and additionally a boolean flag indicating whether the current vertex is an *original vertex* or a *skirt vertex*. An original vertex is a vertex which gets rendered in the exact same location as the terrain mesh's corresponding vertex. A skirt vertex is a vertex whose height will be subtracted by the skirt's height during rendering, thus forming the skirt. The skirt height can be chosen arbitrarily, but usually a value slightly larger than the difference between the minimum and maximum height of the terrain suffices.

In the vertex shader, the scaling, height displacement and ellipsoid transformation occur similarly to the terrain mesh, but as an additional step, the height of each skirt vertex is subtracted by the skirt height. In the fragment shader, the texture coordinate is chosen to be at the exact border of the texture, such that the texture at the border is repeated down the skirt. Figure 4.7 shows an example of a terrain mesh with a skirt mesh.



(a) (b)

Figure 4.7: Illustration of a skirt mesh with original vertices in red and skirt vertices in blue in subfigure (a) and a rendered terrain node with the skirt in subfigure (b).

4.5.3 Pole Mesh

Recall from chapter "Theoretical Background" that the Web Mercator projection cuts off at a latitude of $\pm 85.051129°$. As a result, when rendering the terrain by projecting it to the ellipsoid in the vertex shader, a hole appears where the North and South Pole should be. The solution is to cover up the hole with another special type of mesh: The *pole mesh*.

The pole mesh, similarly to the terrain and the skirt mesh, is a flat mesh centered around $(0, 0, 0)$. The pole mesh has a single vertex is centered at $(0, 0, 0)$ and

n_{pole} vertices along the circle. The vertices form a circle with a radius of 1. The number of vertices along the circle n_{pole} can be programatically defined. StreamingATLOD uses a value of 30. The vertices are generated by splitting the unit circle into n_{pole} pieces using trigonometry:

$$\mathbf{p}_i = (p_{i_x}, p_{i_y}) = (\cos(i \cdot (2\pi/n_{pole})), \sin(i \cdot (2\pi/n_{pole})))$$

for $i \in 0, \ldots, n_{pole} - 1$.

Figure 4.8 shows a wireframe render of the pole mesh.

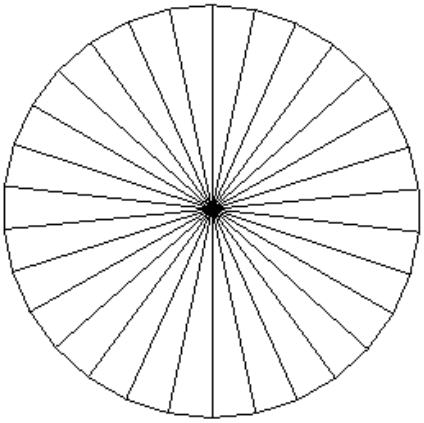


Figure 4.8: Wireframe render of the pole mesh viewed from top.

During rendering, the height of the pole mesh is also displaced in the vertex shader, but now by an uniform `float` variable `earthRadius` instead of with a heightmap, since the height of the pole mesh is the same everywhere. The uniform `float` varaible `isNorthPole` is used as a boolean to negate `earthRadius`, so that both the North and South Pole can be rendered easily. For scaling the pole mesh in the x and z directions, the uniform variable `poleRadius` is used.

In the fragment shader, the pole mesh is rendered with a single color, which is either dark blue for the North Pole, or white for the South Pole, dependent on `isNorthPole`.

4.6 LOD Metric

The *LOD metric* determines at what point a terrain node should be split up into its four child nodes. It gets calculated for every visible node during the collection traversal. If the metric returns true, the traversal should continue with the node's four children, otherwise the traversal ends at the current node.

Recall from subsection "Main Data Structures" that a terrain node stores 9 points in a grid-like manner in the field `_projectedGridPoints`. The LOD

metric function iterates over each point and calculates the distance between the camera's position and the current point, as shown in figure 4.9. If any one of the points lies within a distance threshold, the metric returns true. If all points lie outside of the distance threshold, the metric returns false. and the current node is collected to the render list.

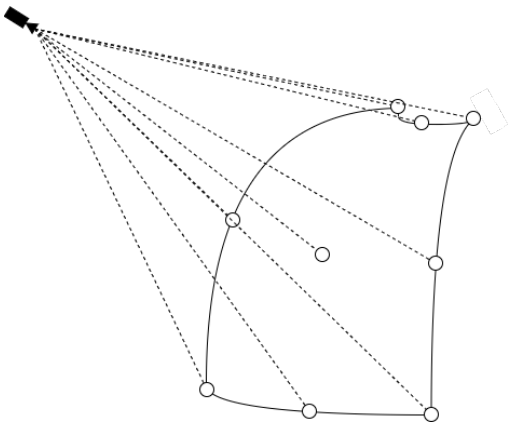


Figure 4.9: Illustration of calculating the distance between the camera and the 9 projected grid points on the tile.

The actual distance threshold is based on a base distance, the current node's zoom level, and the current node's y tile coordinate. The base distance is a predefined constant which represents the distance in world-space units at which the root node gets split up into its four child nodes. The node's zoom level is used to scale the base distance down, thus allowing the same base distance to be used at every zoom level. The node's y tile coordinate is used to suppress the splitting of nodes near the North and South Poles. The reason for this will be made clear in the next few paragraphs.

Recall from chapter "Theoretical Background" that the latitudes converge towards the North and South Poles, as shown in figure 4.10. The terrain nodes gets smaller and more dense towards the poles compared to other nodes with the same zoom level situated closer to the Equator.

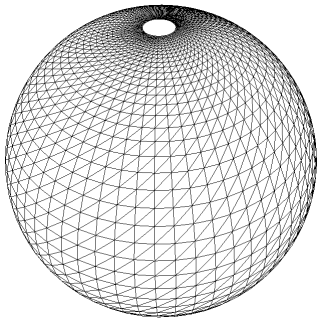


Figure 4.10: Wireframe render of an ellipsoid-projected globe. Notice how the vertices get denser as they approach the North Pole.

Using the same base distance for all latitudes would result in nodes with an unnecessarily high amount of detail being rendered, which would not only put a strain on the rendering performance, since more triangles need to be rendered in a small compressed space, but also on the streaming performance, since more nodes must be loaded than necessary.

The solution to this problem is to reduce the base distance using the y tile coordinate of the current node. The closer a terrain node is to the poles, the lower the base distance will be, therefore requiring the camera to fly closer to the surface in order to stream in and render more detailed nodes near the poles.

In terms of actual values in StreamingATLOD, the base distance is 3.5 times the ellipsoid radius in x direction. Listing 4.3 shows the pseudocode for the split metric. The values were determined through repeated experimentation.

```
1  bool TerrainManager::shouldSplit(Camera& camera, XYZTileKey currentTileKey)
2  {
3      float pow2Level = 1 << currentTileKey.z();
4      TerrainNode* node = _memoryCache.get(currentTileKey.string()).value();
5
6      // Iterate through grid points, check distance
7      for (glm::vec3 point : node->_projectedGridPoints) {
8          glm::vec3 cameraToPoint = point - camera.position();
9          float cameraDistance = glm::length(cameraToPoint);
10
11         // Latitudes close to the poles should have lower base distance
12         float threshold = computeThresholdWithLatitude(currentTileKey);
13
14         // Scale down to appropriate zoom level
15         threshold /= pow2Level;
16
17         if (threshold <= cameraDistance)
18             return true; // Split the terrain node
19     }
20     return false; // Do not split the terrain node
21 }
22
23 float TerrainManager::computeThresholdWithLatitude(XYZTileKey tileKey)
24 {
25     // Use globe radius in x direction times 3.5 as base distance (for root
       node)
26     float threshold = GlobalConstants::GLOBE_RADII.x * 3.5;
27     float pow2Level = 1 << tileKey.z();
28
29     if (tileKey.z() >= 3 && std::abs(tileKey.y() / pow2Level - 0.5f) >= 0.3)
30         threshold *= 0.52;
31     else if (tileKey.y() >= 3 && std::abs(tileKey.y() / pow2Level - 0.5f) >=
       0.4)
32         threshold *= 0.41;
33
34     return threshold;
35 }
```

Listing 4.3: The methods `shouldSplit()` and `computeThresholdWithLatitude()` in `TerrainManager`.

4.7 Culling

Culling refers to the idea of not rendering objects that are not visible to the camera and is of special importance for any rendering system, since it drastically improves the performance by reducing the number of triangles that need to be processed by the GPU and rendered. StreamingATLOD utilizes two culling techniques: *view-frustum culling* and *horizon culling*.

4.7.1 View-frustum Culling

With *view-frustum culling*, objects which lie outside the *view-frustum* do not get rendered. The view-frustum is a three-dimensional pyramid that represents the visible area and is defined by six planes. Figure 4.11 shows an illustration of view-frustum culling with a flat terrain viewed from the top.

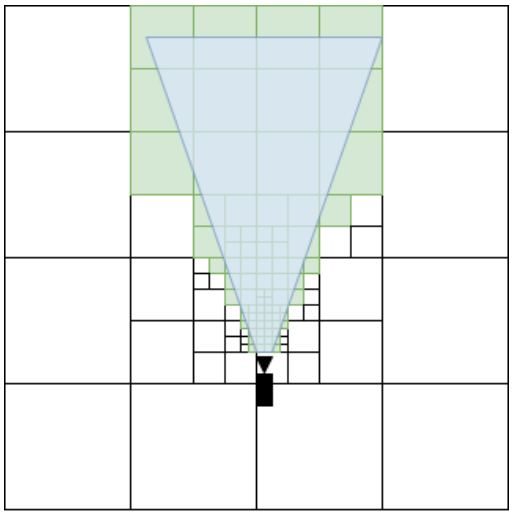
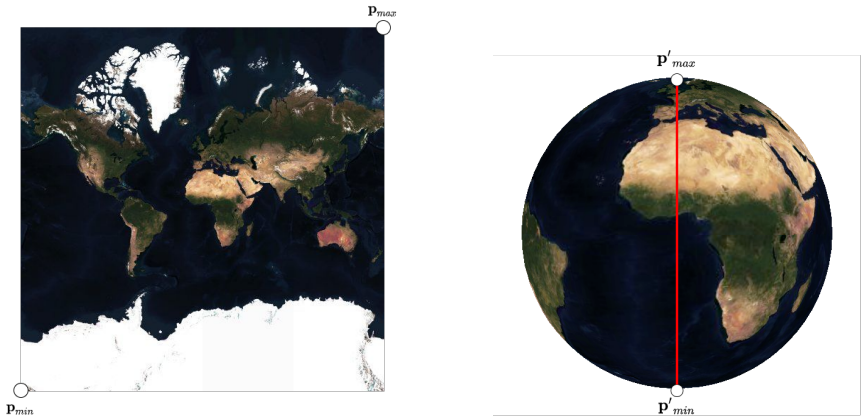


Figure 4.11: Illustration of view-frustum culling with a flat terrain viewed from the top. The blue trapezoid represents the view-frustum and all green nodes are visible and get rendered.

In order to check whether an object, such as a terrain node, is inside the view-frustum, the object needs to have a *bounding volume* associated to it. The bounding volume is simply the volume which contains the entire object inside of it. There are different types of bounding volumes, such as *bounding spheres*, *axis-aligned bounding boxes (AABB)*, *oriented bounding boxes (OBB)*, and more, each with their own strengths and weaknesses.

In StreamingATLOD, the AABB is chosen as the bounding volume for terrain nodes due to its decent balance between accuracy and fast intersection. While bounding spheres are generally faster to generate and intersect, they are significantly more inaccurate, often containing large sections of empty space well above and below the terrain. OBBs can potentially represent a terrain node more accurately but require longer and more complex generation and intersection times.

The usual method of creating a bounding box for a (flat) terrain would be to simply combine the minimum and maximum heights with the two extreme corner points (top-left and bottom-right), $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$. For a non-flat terrain, however, it gets slightly more complicated. For zoom levels 0 and 1, simply projecting $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$ from the flat terrain to the ellipsoid with the geodetic-to-carteisan transformation does not work. Consider for example the level 0 root node. Projecting $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$ results in the new points $\mathbf{p'}_{min}$ and $\mathbf{p'}_{max}$ being located at the North and South Pole respectively. Both projected points have the same longitude, which is $\pm 180°$, and the resulting AABB is a thin plane which stretches from the North to the South Pole. The view-frustum intersection test would yield wrong results in many cases and occlude sections of the Earth when they should be in reality visible. Figure 4.12 visualizes this naive AABB generation method for the root node.



(a) (b)

Figure 4.12: Naive AABB projection to the ellipsoid for the root node. The points $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$ get projected onto the poles as shown in subfigure (b).

To compute the AABB for nodes with zoom level 0, the ellipsoid radii in (x, y, z) directions $\mathbf{r}_{ellipsoid} = (r_x, r_y, r_z)$ is used. For the zoom level 0 tile, the AABB is generated simply by choosing $\mathbf{p}_{min} = -\mathbf{r}_{ellipsoid}$ and $\mathbf{p}_{max} = \mathbf{r}_{ellipsoid}$.

For nodes with zoom level 1, there are four cases: Tile keys $(0, 0, 1)$, $(1, 0, 1)$, $(0, 1, 1)$ and $(1, 1, 1)$. Using once again $\mathbf{r}_{ellipsoid}$, the minimum and maximum points $\mathbf{p}_{min}$ and $\mathbf{p}_{max}$ are computed by taking the ellipsoid radii such that the entire section of the Earth represented by that node lies inside of the two points. For example, a node with the tile key $(1, 0, 1)$ (top-right section of the Earth at zoom level 1) gets an AABB with the points $\mathbf{p}_{min} = (0, 0, -r_z)$ and $\mathbf{p}_{max} = (r_x, r_y, r_z)$.

For nodes with higher zoom levels, the four corner points of the node can now be projected onto the ellipsoid and the minimum and maximum (x, y, z) coordinates taken from the four projected points. Each node with zoom level 2 or higher has a minimum and maximum longitude lon_{min} and lon_{max} such that $|lon_{max} - lon_{min}| \leq 90°$. This means that for nodes with a zoom level 2 or higher, the four corner points all lie within the same quarter of any hemisphere,

which was not the case for zoom levels 0 and 1. Figure 4.13 shows an illustration of an AABB encasing an ellipsoid-projected terrain node.

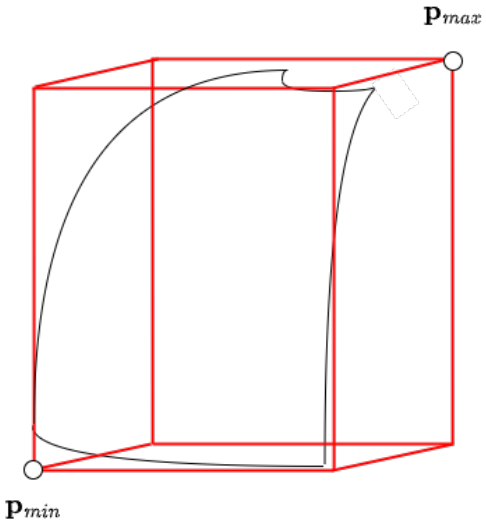


Figure 4.13: Illustration of an ellipsoid-projected terrain node with its AABB in red.

StreamingATLOD's view-frustum culling implementation is based on chapter "Frustum Culling" in *Learn OpenGL* by de Vries [dV20]. The methods used for view-frustum culling are implemented in the `Camera` class and shown in listing 4.4.

```
1  bool Camera::insideViewFrustum(glm::vec3 p1, glm::vec3 p2)
2  {
3      Frustum frustum = _viewFrustum;
4
5      unsigned checked = 0;
6
7      checked += checkPlane(frustum.leftFace, p1, p2);
8      checked += checkPlane(frustum.rightFace, p1, p2);
9      checked += checkPlane(frustum.topFace, p1, p2);
10     checked += checkPlane(frustum.bottomFace, p1, p2);
11     checked += checkPlane(frustum.nearFace, p1, p2);
12     checked += checkPlane(frustum.farFace, p1, p2);
13
14     return checked == 6;
15 }
16
17 bool Camera::checkPlane(Plane& plane, glm::vec3 p1, glm::vec3 p2)
18 {
19     float minY = p1.y;
20     float maxY = p2.y;
21     float width = p2.x - p1.x;
22     float depth = p2.z - p1.z;
23     glm::vec3 aabbCenter = p1 + ((p2 - p1) / 2.0f);
24
25     float halfHeight = (maxY - minY) / 2.0f;
26     float halfWidth = width / 2.0f;
27     float halfDepth = depth / 2.0f;
28     float r = halfWidth * std::abs(plane.normal.x)
29         + halfHeight * std::abs(plane.normal.y)
30         + halfDepth * std::abs(plane.normal.z);
```

```
31
32      return -r <= plane.getSignedDistanceToPlane(aabbCenter);
33  }
```

Listing 4.4: The methods `Camera::insideViewFrustum` and `Camera::checkPlane` for view-frustum culling.

4.7.2 Horizon Culling

View-frustum culling eliminates a large number of nodes which are not visible to the camera, but not all of them. Consider the following scenario. The camera is above Singapore and looking straight into the ground towards the center of the Earth. Intuitively, only the area around Singapore should be rendered. However, the view-frustum also (correctly) intersects with the AABB containing the majority of South America, which lies on the opposite side of the Earth, resulting in a large section of the Earth being rendered despite not even being visible. *Horizon culling* solves this problem by omitting nodes which lie beyond the visible horizon from being rendered, as shown in figure 4.14.

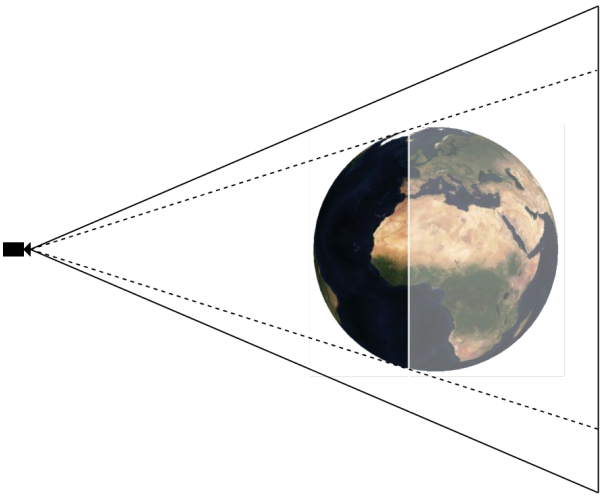


Figure 4.14: Illustration of horizon culling. The solid lines represents the view-frustum and the dotted lines are the half-open line segments starting from the camera's position going through the horizon points.

StreamingATLOD's implementation of horizon culling is largely based on the technique described in the blog article written by Ring at Cesium Geospatial [Rin13], which includes a derivation of the formulas. The points for checking against the horizon `_horizonCullingPoints` are calculated during the creation of the `TerrainNode*` instance using the node's heightmap, in a similar way to the nine `_projectedGridPoints`. The height of each point is set to be slightly higher than the actual height value from the heightmap in order to avoid the visual artifact of the horizon suddenly popping into view in the distance. Listing 4.5 shows the pseudocode for checking whether a terrain node is horizon

culled.

```
bool TerrainNode::horizonCulled(Camera& camera)
{
    glm::vec3 cv = camera.position() / GlobalConstants::GLOBE_RADII;

    float vhMagnitudeSquared = glm::dot(cv, cv) - 1.0f;

    // Check if all 9 points lie beyond the horizon
    for (auto p : _horizonCullingPoints) {
        glm::vec3 t = p / globeRadius;
        glm::vec3 vt = t - cv;
        float vtMagnitudeSquared = glm::dot(vt, vt);
        float vtDotVc = -1.0f * glm::dot(vt, cv);
        bool isOccluded = vtDotVc > vhMagnitudeSquared && vtDotVc * vtDotVc /
     vtMagnitudeSquared > vhMagnitudeSquared;

        if (!isOccluded)
            return false;
    }
    return true;
}
```

Listing 4.5: Method `horizonCulled()` in `TerrainNode`.

4.8 Terrain Caching

This section describes the caching mechanisms for terrain data. As previously mentioned in section "System Overview", StreamingATLOD uses a *disk cache* and a *memory cache*. Both caches are managed as LRU caches with fixed capacities. Reflected by the fact that computers usually have a much larger disk capacity than memory, StreamingATLOD's disk cache disk cache is usually much larger than the memory cache. The cache sizes can be set in the configuration file with the requirement that the disk cache is at least 4 times larger than the memory cache.

In addition to LRU, both caches use additional special *eviction policies* which specify certain special cases for when a least-recently used entry should or should not be evicted.

4.8.1 Disk Cache

The disk cache caches terrain data on the disk in order to lower the number of API requests. The disk cache consists of the *in-memory disk cache* and the *file system disk cache*. The in-memory disk cache stores tile keys of terrain nodes whose data is cacjed in the file system disk cache. The file system disk cache is a folder on the client's file system that contains the actual heightmap tiles and overlay tiles.

The in-memory disk cache is a field of `TerrainManager` named `_diskCache` and is of the type `LRUCache<std::string, void*>`. The value type is `void*`, since the only required information is already in the key.

Structure of the Disk Cache

As previously mentioned, the file system disk cache is a folder specified by the user on the client's file system. The folder has two subfolders: the folder `heightdata`, which stores heightmap tiles, and the folder `overlay`, which stores overlay imagery tiles. The tiles of both types are stored with the following file name convention: `x_y_z.ext`, where `x`, `y` and `z` are the XYZ tile coordinates and `.ext` is the file extension (`.jpg` for overlay tiles and `.webp` for heightmap tiles).

Initialization

StreamingATLOD needs to know which terrain nodes are already cached on the disk. The method `TerrainManager::initDiskCache()`, which is executed at start up, traverses the file system disk cache and collects all file names of the tiles in the file system disk cache. If there is a heightmap tile that does not have a corresponding overlay imagery tile or vice-versa, it gets deleted. Finally, the XYZ tile key of each collected tile gets put into the in-memory disk cache. If the cache's capacity is reached during start up, the tiles are deleted according to the same eviction policy used at runtime (see below).

Eviction Policy

The overlay and heightmap with a given XYZ tile key cannot get evicted from the disk cache if at least one of the following is true:

- It is currently being loaded (i.e. the tile key is in `_loadingNodes`).
- The current node has at least one child that is cached on the disk as well (i.e. at least one child tile key is in `_diskCache`).
- The terrain node is the root node.

The reason for these limitations is that we always want to evict from bottom (high zoom levels) to top (low zoom levels) if possible.

Putting and Evicting Data at Runtime

Whenever we process a newly loaded terrain node from the load worker done queue `_loadDoneQueue` in the method `processAllDoneQueue()` in the main thread, we initialize the terrain node in `initTerrainNode()`. In `initTerrainNode()` we also need to insert the tile key of the newly loaded node into the in-memory disk cache if it isn't inside of it already and potentially evict an old disk cache entry if the disk cache is full. Listing 4.6 shows the process of doing so inside `initTerrainNode()`.

```
1 void TerrainManager::initTerrainNode() {
2     // Initialize node ...
3     // Insert into memory cache ...
4     // ...
5
6     PutResult<XYZTileKey, void*> diskResult = _diskCache.put(tile->xyzTileKey
      (), nullptr);
7     if (diskResult.evicted) {
8         XYZTileKey evictedKey = diskResult.evictedItem.value().first;
```

```
9           // Take the next evicted entry while it does not adhere to the
        eviction policy
10          while (!checkEviction(evictedKey, nullptr) || _loadingTiles.count(
        evictedKey)) {
11              diskResult = _diskCache.put(evictedKey, nullptr);
12              evictedKey = diskResult.evictedItem.value().first;
13          }
14          _currentDiskCacheEvictions.insert(evictedKey);
15          _unloadRequestQueue->push({ evictedKey, UNLOAD_REQUEST });
16      }
17  }
```

Listing 4.6: Inserting a new in-memory disk cache entry with a potential eviction.

There are four possible cases:

- The disk cache is not full and `_diskCache` does not contain the tile key of the newly loaded terrain node yet. In this case, when putting the tile key in the disk cache, the tile key gets pushed to the front of the in-memory disk cache, making it the most recently-used entry. The `PutResult`'s `eviction` flag is set to false in line 7 and disk cache entry gets evicted.
- The disk cache is not full and `_diskCache` already contains the tile key of the newly loaded terrain node. In this case, we proceed indentically as above, except that the cache does not have any new entries.
- The disk cache is full and `_diskCache` contains the tile key for the newly loaded terrain node. Here, we proceed similarly to the previous point, in which the already existing tile key gets pushed to the front without an eviction occurring.
- The disk cache is full and `_diskCache` does not contain the tile key for the newly loaded terrain node. This case is special, since an eviction occurs which needs to be handled. We first check whether we are allowed to evict the entry by entering the while loop condition check in line 10. If the entry can be evicted, we skip the while loop's body, track the tile key as being currently evicted in `_currentDiskCacheEvictions` and proceed to send a request to the disk deallocation worker request queue (lines 14 and 15).

The while loop (lines 10 to 13) may seem problematic at first, since it bypasses the LRU policy by reinserting a least-recently used item back to the front of the cache. Also, at first sight there might seem like a risk of long waiting times in the while loop if there are many entries towards the end of the cache that cannot be evicted. However, in practice, these points are not as problematic as they seem. When a least-recently used entry is checked for whether it can be evicted and it turns out it cannot be, it makes the most sense to put it in the front of the cache, since want to check all other nodes for eviction. Also, the problem of long waiting times in the while loop is unlikely. The collection traversal occurs top-down (from low zoom levels to high zoom levels) and when the camera flies from one place to another, terrain nodes further up the in the quadtree are likely traversed again throughout subsequent render frames, as shown in figure 4.15. As a result, the entries towards the end of the cache tend to be nodes with high zoom levels with likely few child nodes, if at all.

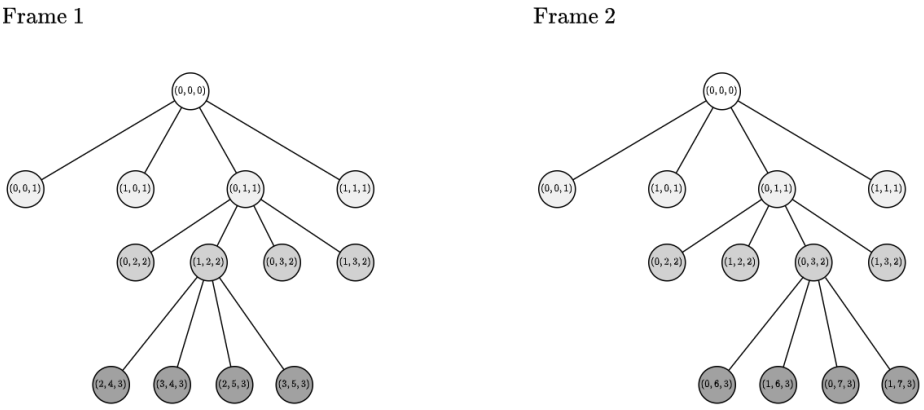


Figure 4.15: Illustration of a terrain node tree in a first and a second render frame. The nodes further up in the hierarchy are more likely to be used in throughout subsequent frames than nodes further down.

Disk Deallocation Worker

The disk deallcation worker is the worker thread that performs the actual deletion of terrain tiles on the file system disk cache and is represented by the class `DiskDeallocationWorkerThread`. The `TerrainManager` manages it and stores a `DiskDeallocationWorkerThread _unloadWorkerThread`, a `MessageQueue<DiskDeallocationRequest>* _unloadRequestQueue` for sending requests , `MessageQueue<DiskDeallocationResponse>* _unloadDoneQueue` for receiving responses and a `std::unordered_set<XYZTileKey> _currentDiskCacheEvictions` for tracking entries that are currently being deallocated from the disk.

4.8.2 Memory Cache

The memory cache is the main storage for terrain nodes which are currently loaded in memory and ready to be rendered. The memory cache is stored as the field `LRUCache<std::string, TerrainNode*> _memoryCache` of the `TerrainManager` class.

Eviction Policy

The additional eviction policies for the memory cache are almost identical to the disk cache, except we do not check whether the data is currently being loaded, since it already is loaded in memory.

Putting and Evicting Data at Runtime

The eviction happens almost identically to the disk cache's eviction shown in listing 4.6, except that we destroy the `TerrainNode*` instance and its associated OpenGL texture objects directly in the main thread, rather than on a separate deallocation thread.

4.9 Terrain Streaming

This section dives into the multithreaded streaming mechanism of StreamingATLOD. For this, the section will mainly resolve around the load worker thread, which is responsible for loading data in the background.

4.9.1 Load Worker Thread

The load worker thread is encapsulated in the class `LoadWorkerThread`, as shown in listing 4.7.

```
1  class LoadWorkerThread
2  {
3  public:
4      LoadWorkerThread(MessageQueue<LoadRequest>* requestQueue, MessageQueue<
       LoadResponse>* doneQueue);
5
6      void postRequest(XYZTileKey tileKey);
7      void startInAnotherThread();
8      void run();
9
10     void processAllRequests();
11     void fetchOverlay(LoadRequest& request, LoadResponse& response);
12     void fetchHeightmap(LoadRequest& request, LoadResponse& response);
13
14     void loadHeightmapFromDisk(LoadRequest& request, LoadResponse& response);
15     void loadOverlayFromDisk(LoadRequest& request, LoadResponse& response);
16     void loadHeightmapFromApi(LoadRequest& request, LoadResponse& response);
17     void loadOverlayFromApi(LoadRequest& request, LoadResponse& response);
18
19     MessageQueue<LoadRequest>* _requestQueue;
20     MessageQueue<LoadResponse>* _doneQueue;
21
22     std::thread _thread;
23     CURL* _curl;
24     bool _stopThread = false;
25 };
```

Listing 4.7: The class `LoadWorkerThread`.

Its members are the request queue and the done queue, the `std::thread` instance, the `CURL` handle for loading tiles from the web APIs, and the boolean flag that indicates whether the thread should be stopped. The cURL handle is defined as a member in order to be able to reuse it over multiple requests, which improves performance [ec].

Load Request and Response Types

The load worker request and done queues hold values of the struct types `LoadWorkerRequest` and `LoadWorkerResponse` respectively.

The struct `LoadWorkerRequest`, shown in listing 4.8 contains the tile key that should be loaded and an enum `LoadRequestType` indicating whether to load from the disk cache or from the web API.

```
1  enum LoadRequestType {
2      LOAD_REQUEST,
3      LOAD_REQUEST_DISK_CACHE,
```

```
4       LOAD_STOP_THREAD
5   };
6
7   struct LoadRequest {
8       XYZTileKey tileKey;
9       LoadRequestType type;
10  };
```

Listing 4.8: The struct `LoadRequest` and enum `LoadRequestType`.

The struct `LoadWorkerResponse`, shown in listing 4.9 contains the enum `LoadResponseType` as a return code for the main thread, the newly created `TerrainTile*`, heap-allocated `unsigned char` pointers to the raw heightmap and overlay texture and their image dimensions, and an enum `LoadResponseOrigin` indicating whether the data was loaded from the disk cache or from the web API. The raw data pointers will be deallocated in the main thread when they are not needed anymore.

```
1   enum LoadResponseType {
2       LOAD_OK, // Loaded alright
3       LOAD_ERROR, // Network error
4       LOAD_UNLOADABLE, // No error, but tile does not exist, e.g. oceans at
        high zoom levels
5       LOAD_TIMEOUT, // Web API timed out due to e.g. rate limiting
6       LOAD_STOPPED_THREAD // Signal for stopping the worker thread
7   };
8
9   enum LoadResponseOrigin {
10      LOAD_ORIGIN_DISK_CACHE,
11      LOAD_ORIGIN_API
12  };
13
14  struct LoadResponse {
15      LoadResponseType type;
16      TerrainNode* tile;
17      unsigned char* heightData;
18      unsigned char* overlayData;
19      unsigned heightmapWidth;
20      unsigned heightmapHeight;
21      unsigned overlayWidth;
22      unsigned overlayHeight;
23      LoadResponseOrigin origin;
24  };
```

Listing 4.9: The struct `LoadResponse`, enum `LoadResponseOrigin` and enum `LoadResponseType`.

4.9.2 Load Worker Request Queue Processing

The method `startInAnotherThread()` starts the load worker thread in another thread and starts the listening loop, which continually calls the method `processAllRequests()` until the thread should be stopped. The method `processAllRequests()` is responsible for processing messages coming in on the request queue and is shown in listing 4.10.

```
1   void LoadWorkerThread::processAllRequests()
2   {
```

```
3       auto requests = _requestQueue->popAll();
4
5       bool returnNetworkErrors = false;
6
7       for (auto request : requests) {
8           if (request.type == LOAD_STOP_THREAD)
9               _stopThread = true;
10
11          LoadResponse response = { LOAD_OK, request.tileKey, nullptr, nullptr,
          nullptr, 0, 0, 0, 0, 0, LOAD_ORIGIN_DISK_CACHE };
12
13          if (returnNetworkErrors) {
14              response.type = LOAD_ERROR;
15              _doneQueue->push(response);
16              continue;
17          }
18
19          fetchHeightmap(request, response);
20
21          if (response.type != LOAD_OK) {
22              if (response.type == LOAD_ERROR)
23                  returnNetworkErrors = true;
24              _doneQueue->push(response);
25              continue;
26          }
27
28          fetchOverlay(request, response);
29
30          if (response.type != LOAD_OK) {
31              if (response.type == LOAD_ERROR)
32                  returnNetworkErrors = true;
33              _doneQueue->push(response);
34              continue;
35          }
36
37          if (response.type == LOAD_OK) {
38              TerrainNode* newTile = new TerrainNode(request.tileKey);
39              newTile->_heightData = response.heightData;
40
41              newTile->generateMinMaxHeight();
42              newTile->generateAabb();
43              newTile->generateProjectedGridPoints();
44              newTile->generateHorizonPoints();
45
46              response.tile = newTile;
47          }
48
49          _doneQueue->push(response);
50      }
51
52      // Respond to main thread that we have stopped the thread
53      if (_stopThread)
54          _doneQueue->push({ LOAD_THREAD_STOPPED, request.tileKey, nullptr,
         nullptr, nullptr, 0, 0, 0, 0, 0, LOAD_ORIGIN_DISK_CACHE });
55  }
```

Listing 4.10: The method `LoadWorkerThread::processAllRequests()`.

When processing the request queue, the method first pops all messages off the request queue (line 3) and loads first the heightmap (line 19) and then the overlay (line 28). If the heightmap failed loading, then we preemptively send back a response with a failure response type without loading the overlay texture (lines 13 to 17), since we cannot render a terrain node if at least one of them is unavailable. The same occurs if the heightmap succeeded loading but the overlay failed (lines 30 to 36). The reason for sequentially loading the heightmap and the overlay image is to reduce the number of requests in case of an error or an unloadable tile, since we preemptively stop processing the request as soon as

the heightmap loading fails.

If the response type is `LOAD_UNAVAILABLE` or `LOAD_TIMEOUT`, then we simply send back the response with that code. If the response type is `LOAD_ERROR`, then something must be wrong with the network connection, which we have to handle specially. We first set `returnNetworkErrors` to true (lines 23 and 32), which causes all subsequent requests to be responded to with responses of the type `LOAD_ERROR` (lines 13 to 17). These responses will signal to the main thread that it should wait for a short amount of time before sending new web API requests.

If the main thread signalled to the load worker that it should stop running by sending a request of the type `LOAD_STOP_THREAD`, for example when the application shuts down, the load worker sets its member flag `_stopThread` to true (line 8 and 9) and finishes procesing all remaining requests, after which it sends a response of the main thread of the type `LOAD_THREAD_STOPPED`.

4.9.3 Overlay and Heightmap Loading

The methods `fetchOverlay()` and `fetchHeightmap()` are responsible for loading the overlay and heightmap of a terrain node respectively. Both functions take mutable references to the requests and responses defined in the method `processAllRequests()`. At the end of both function calls, the `response` variable in the caller should contain all relevant information, such as terrain data, response code and the origin of the response (disk or API). Both functions are shown in listing 4.11.

```
1  void LoadWorkerThread::fetchHeightmap(LoadRequest& request, LoadResponse&
       response)
2  {
3      if (request.type == LOAD_REQUEST_DISK_CACHE) { // Load from disk
4          loadHeightmapFromDisk(request, response);
5          response.origin = LOAD_ORIGIN_DISK_CACHE;
6      } else if (!request.offlineMode) { // Load from API if not offline
7          loadHeightmapFromApi(request, response);
8          response.origin = LOAD_ORIGIN_API;
9      } else // Offline mode, we cannot request new tiles
10         response.type = LOAD_UNLOADABLE;
11 }
12
13 LoadWorkerThread::fetchOverlay(LoadRequest& request, LoadResponse& response)
14 {
15     if (request.type == LOAD_REQUEST_DISK_CACHE) {
16         loadOverlayFromDisk(request, response);
17         response.origin = LOAD_ORIGIN_DISK_CACHE;
18     } else if (!request.offlineMode) {
19         loadOverlayFromApi(request, response);
20         response.origin = LOAD_ORIGIN_API;
21     } else // Offline mode, we cannot request new tiles
22         response.type = LOAD_UNLOADABLE;
23 }
```

Listing 4.11: The methods `loadOverlay()` and `loadHeightmap()`.

Both functions work very similarly. If the request for the terrain data is of the type `LOAD_REQUEST_DISK_CACHE`, then it means that the in-memory disk cache in the main thread contains the required entry and we can load the data from

the file system disk cache into memory in the load worker thread.

4.9.4 Loading Data from the Disk Cache

The process for loading the overlay texture and the heightmap texture is essentially identical, except for the fact that the overlay image format is JPG and that the heightmap image format is WebP. For this reason, in this and the next subsections discussing the loading process, we will only look at the loading process for the overlay texture in order to avoid redundant explanations.

The method `loadOverlayFromDisk()` simply loads the overlay image from the file system disk cache using the disk cache path specified in the configuration file (see Section "Additional Features"). This method should not be able to fail, since the in-memory disk cache indicated that the overlay image is in the file system disk cache. Inn the exceptional case that it does fail, we simply mark it as unloadable. Listing 4.12 shows the method `loadOverlayFromDisk()`.

```
1  void LoadWorkerThread::loadOverlayFromDisk(LoadRequest& request, LoadResponse
       & response)
2  {
3      XYZTileKey tileKey = request.tileKey;
4      int width, height, nrChannels;
5      std::string fileName = ConfigManager::getInstance()->diskCachePath()
6          + "overlay/" + std::to_string(tileKey.x())
7          + "_" + std::to_string(tileKey.y())
8          + "_" + std::to_string(tileKey.z()) + ".jpg";
9
10     unsigned char* data = stbi_load(fileName.c_str(), &width, &height, &
       nrChannels, 0);
11
12     if (data) {
13         response.overlayData = data;
14         response.overlayWidth = width;
15         response.overlayHeight = height;
16         response.type = LOAD_OK;
17     } else {
18         std::cerr << "Failed opening overlay texture from cache " << tileKey.
       string() << std::endl;
19         response.type = LOAD_UNLOADABLE;
20     }
21 }
```

Listing 4.12: The method `LoadWorkerThread::loadOverlayFromDisk`.

4.9.5 Loading Data from the Web API

The method `loadOverlayFromApi()` loads the overlay and is shown in listing 4.13.

```
1  void LoadWorkerThread::loadOverlayFromApi(LoadRequest& request, LoadResponse&
       response)
2  {
3      XYZTileKey tileKey = request.tileKey;
4
5      std::string responseData;
6      std::string url = ConfigManager::getInstance()->overlayDataServiceUrl()
7          + std::to_string(tileKey.z()) + "/"
8          + std::to_string(tileKey.x()) + "/"
```

```
9           + std::to_string(tileKey.y()) + ".jpg?key=" + ConfigManager::
        getInstance()->overlayDataServiceKey();
10
11      if (!_curl) { // Curl handle failed
12          std::cerr << "Curl failed" << std::endl;
13          std::exit(1);
14      }
15
16      curl_easy_setopt(_curl, CURLOPT_URL, url.c_str());
17      curl_easy_setopt(_curl, CURLOPT_WRITEFUNCTION, writeData);
18      curl_easy_setopt(_curl, CURLOPT_WRITEDATA, &responseData);
19      curl_easy_setopt(_curl, CURLOPT_TIMEOUT, 5L); // Timeout is after 5
        seconds
20
21      int httpStatusCode = 0;
22      int retCode = curl_easy_perform(_curl);
23
24      // Timeout occurred
25      if (retCode == CURLE_OPERATION_TIMEDOUT) {
26          std::cout << "Heightmap timeout" << std::endl;
27          response.type = LOAD_TIMEOUT;
28          return;
29      }
30
31      // Get HTTP status code
32      curl_easy_getinfo(_curl, CURLINFO_RESPONSE_CODE, &httpStatusCode);
33
34      if (retCode != CURLE_OK) { // Network error
35          std::cerr << "Network error" << std::endl;
36          response.type = LOAD_ERROR;
37          return;
38      } else if (httpStatusCode == 204) { // Unloadable tile, e.g. oceans at
        high zoom levels
39          std::cerr << "Tile is empty" << std::endl;
40          response.type = LOAD_UNLOADABLE;
41          return;
42      } else if (httpStatusCode != 200) { // Something else went wrong in the
        API
43          std::cerr << "Status code " << httpStatusCode << " in loading overlay
        " << std::endl;
44          response.type = LOAD_ERROR;
45          return;
46      }
47
48      int width, height, nrChannels;
49      unsigned char* data = stbi_load_from_memory((unsigned char*)responseData.
        c_str(), responseData.size(), &width, &height, &nrChannels, 0);
50      if (data) {
51          std::string filePath = ConfigManager::getInstance()->diskCachePath()
        + "overlay/"
52              + std::to_string(tileKey.x()) + "_"
53              + std::to_string(tileKey.y()) + "_"
54              + std::to_string(tileKey.z()) + ".jpg";
55          std::ofstream file(filePath, std::ios::out | std::ios::binary);
56          if (file.is_open()) {
57              file.write(responseData.c_str(), responseData.size());
58              file.close();
59          } else { // Couldn't open new file for texture image, something
        severly went wrong
60              std::cerr << "Failed to open file for writing." << std::endl;
61              std::exit(1);
62          }
63
64          response.overlayData = data;
65          response.overlayWidth = width;
66          response.overlayHeight = height;
67          response.type = LOAD_OK;
68      } else { // Couldn't decode overlay texture image, something severly went
          wrong
69          std::cerr << "Failed opening overlay texture" << std::endl;
70          std::exit(1);
71      }
72  }
```

Listing 4.13: The method `loadOverlayFromApi()`.

As a first step, it builds the request URL using the URL and API key set in the configuration (lines 3 to 9). Afterwards, it checks whether the cURL handle is valid (lines 11 to 14) and if so, sets the request options, such as the URL, the write function, the response buffer and the maximum time before timeout (lines 16 to 19). Then, it performs the actual cURL HTTP request (line 22). First, we check whether the HTTP request timed out and if so, we set our response to `LOAD_ERROR` and return (lines 24 to 29). Otherwise, we retrieve the HTTP status code (line 32) and check for various conditions, such as whether a network error occurred (lines 34 to 37), the tile is unloadable (lines 38 to 41), or whether the web API responded with a HTTP status code $\neq 200$ (lines 42 to 45). If none of these conditions were true, we load decode the overlay image using `stbi_load_from_memory()` (line 49) and store the image to the file system disk cache (lines 51 to 58). As a final step, we set the response struct members to contain the raw decoded overlay image data, the width and height of the overlay image and set the response type to `LOAD_OK` (lines 64 to 67).

4.9.6 Processing New Nodes in the Render Thread

As previously mentioned, the render thread performs two main operations of collecting the terrain nodes to be rendered and rendering them. But before performing these two operations, it processes new messages from the various worker thread queues. The method `TerrainManager::processAllLoadDoneQueue()` retrieves and processes all elements from the load worker done queue and is shown in listing 4.14.

```
1  void TerrainManager::processAllDoneQueue()
2  {
3      std::deque<LoadResponse> responses = _doneQueue->popAll();
4
5      for (auto response : responses) {
6          // Finished loading, do not need to track anymore
7          _loadingNodes.erase(response.tileKey);
8
9          // Handle potential errors or unloadable tiles
10         if (response.type == LOAD_UNLOADABLE || response.type == LOAD_TIMEOUT
       || response.type == LOAD_ERROR) {
11             // Unloadable node, store it in unloadable set to prevent future
       requests
12             if (response.type == LOAD_UNLOADABLE)
13                 _unloadableTileKeys.insert(response.tileKey);
14
15             // Network error, wait for 5 seconds before starting new request
16             if (response.type == LOAD_ERROR) {
17                 _lastNetworkError = std::chrono::system_clock::now();
18                 _offlineWait = true;
19             }
20
21             if (response.heightData != nullptr) {
22                 delete[] response.heightData;
23             }
24             if (response.overlayData != nullptr) {
25                 stbi_image_free(response.overlayData);
26             }
27
```

```
28              delete response.node;
29              continue;
30          } else
31              initTerrainNode(response);
32      }
33  }
```

Listing 4.14: The method `LoadWorkerThread::processAllDoneQueue()`.

We first pop all entries from the load worker done queue (line 3). Afterwards, we iterate all popped entries and first remove the entry of the tile key from the set of tile keys whose nodes are being loaded (line 7). Next, we check whether we received an error or an unloadable node (line 10) and check the following two cases:

- If the node is unloadable, then we put it in the set of unloadable nodes `_unloadableTileKeys` (lines 12 and 13) to avoid requesting it again in the future.
- If the response was an error, we update the timestamp that indicates when the last error occurred `_lastNetworkError` to the current time (line 17) and and set the flag `_offlineWait` to true (line 18), indicating that we should wait a few seconds before requesting new data from the web APIs.

Finally, we delete the data of the terrain node (lines 21 to 28) and continue with the next entry (line 29).

If no error occured, we simply continue processing the terrain node in `TerrainManager::initTerrainNode()`.

In it, we generate the OpenGL texture objects for the heightmap and overlay that we just received and put the node's tile key in the memory cache and the disk cache if it was loaded from the web API.

The overlay textures additionally get generated with mipmaps for trilinear filtering in order to reduce aliasing artifacts in the distance. Since the textures always have a width and height of 512 pixels, which is a power of two, mipmap generation occurs quickly and should not significantly hinder rendering performance. It also comes with an additional GPU memory overhead of only 33%. Both textures are clamped in order to avoid texture artifacts due to limited texture coordinate precision at the border of the textures. Listing 4.15 shows the OpenGL texture generation pseudocode.

```
1  void TerrainManager::initTerrainNode(LoadResponse response) {
2      TerrainNode* node = response.node;
3
4      // Overlay texture
5      glGenTextures(1, &node->overlayTextureId);
6      glBindTexture(GL_TEXTURE_2D, node->overlayTextureId);
7
8      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
9      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
10     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
       GL_LINEAR_MIPMAP_LINEAR);
11     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
12
13     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, overlayWidth, overlayHeight, 0,
       GL_RGB, GL_UNSIGNED_BYTE, overlayData);
```

```
14      glGenerateMipmap(GL_TEXTURE_2D);
15
16      // Heightmap texture
17      glGenTextures(1, &node->heightmapTextureId);
18      glBindTexture(GL_TEXTURE_2D, node->heightmapTextureId);
19
20      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
21      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
22      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
23      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
24
25      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, heightmapWidth, heightmapHeight,
         0, GL_RGB, GL_UNSIGNED_BYTE, heightmapData);
26
27      // Continue with putting into LRU caches, eviction checks, etc.
28      // ...
29  }
```

Listing 4.15: OpenGL texture generation for the overlay and the heightmap in `initTerrainNode()`.

4.10 Rendering Process

At last, we arrive at the actual process of rendering the terrain. In this section, the main operations of what happens during a single render frame are described. They are encapsulated by the method `TerrainManager::render()`, which is shown in listing 4.27.

```
1  void TerrainManager::render(Camera& camera)
2  {
3      // Process newly requested nodes that came back from the worker threads
4      processAllDoneQueue();
5      processAllUnloadDoneQueue();
6
7      std::queue<XYZTileKey> visibleNodes;
8
9      // Collect starting from the root node
10     collectVisible(camera, XYZTileKey(0, 0, 0), visibleNodes);
11
12     // Render all collected nodes
13     renderVisible(camera, visibleNodes);
14 }
```

Listing 4.16: Method `TerrainManager::render()` that gets called every frame to render the terrain.

First, we process all responses from the load worker response queue and the disk deallocation response queue (lines 4 and 5). Afterwards, we collect the visible nodes starting from the root node (lines 7 and 10) and subsequently render all collected nodes (line 13).

4.10.1 Collecting Visible Nodes

The collection of visible nodes is the central operation of the LOD algorithm, since this is where the quadtree traversal, the culling, the LOD metric calculation and the requesting for unloaded nodes happen. This is the exact place

where all the concepts of the previously described sections come together. Collecting visible nodes is a recursive operation and requires traversing through the terrain nodes starting from the root node. The recursive traversal has two base cases:

1. The current terrain node is not visible.
2. The current terrain node should be rendered.

These two base cases stop the recursive traversal for the subtree whose root is the current node.

The following steps get performed per terrain node:

- First we check whether the terrain is visible at all by performing view-frustum culling and horizon culling. Horizon culling only gets performed if the node is inside the view-frustum, since view-frustum culling is a stricter requirement; If a terrain node is outside the view-frustum, it is definitely not visible. If a terrain node is inside the view-frustum, it can be visible, but does not have to be. If the current terrain node is not visible, we stop the traversal at the current node. Note that the current node's siblings or ancestors might still get traversed afterwards, since this is a depth-first traversal. Otherwise, if the current terrain node is visible, we continue with the next operations explained below.
- Next we check whether the current terrain node should be split up into its four child nodes. For this, we use the LOD metric, which returns true if the terrain node should be split up, false otherwise. If the terrain node should not be split up, we add it to the render list and stop the traversal at the current node. Otherwise, if the terrain node should be split up, we have three possible cases:
  - All four child nodes are loaded. In this case, we recursively call the traversal algorithm on the child nodes and repeat the just described steps.
  - At least one of the child nodes is not loaded. In this case, we add the current node to the render list, request the missing child nodes to be loaded in and stop the traversal at the current node.
  - At least one of the child nodes cannot be loaded. Recall from section "Terrain Streaming" that not the entire world is available up to the maximum zoom level, for example at oceans, since there is nothing of interest to see there. In such a case, a libcurl request to the web API returns the HTTP status code 204, indicating that there is no data for the sent tile key. In such a case, we simply add the current node to the render list and end the traversal at the current node.

The collection traversal is encapsulated by the method `collectVisible()`, which is shown in pseudocode in listing 4.17.

```
1 void TerrainManager::collectVisible(Camera& camera, XYZTileKey currentTileKey
      , std::queue<XYZTileKey>& visibleNodes, XYZTileKey&
      minimumDistanceTileKey)
2 {
```

```
3       // Put current node to front of disk cache (if existant)
4       _diskCache.get(currentTileKey);
5
6       TerrainNode* node = _memoryCache.get(currentTileKey).value();
7       node->_lastUsedTimeStamp = std::chrono::system_clock::now();
8
9       unsigned level = currentTileKey.z();
10
11      bool visible = camera.insideViewFrustum(node->aabbP1(), node->aabbP2());
12
13      // We do horizon culling only from level 2 upwards
14      if (level >= 2)
15          visible = visible && !currentTile->horizonCulled(camera);
16
17      // Not visible, we are done here
18      if (!visible)
19          return;
20
21      bool split = shouldSplit(camera, currentTileKey) && level < _maxZoom;
22
23      if (!split) {
24          // We want to render the current node
25          updateMinimumDistanceTileKey(camera, currentTileKey,
        minimumDistanceTileKey);
26          visibleNodes.push(node);
27      } else {
28          if (!allChildrenExistant(currentTileKey)) {
29              // We have to render the current node because the children are
        not loaded yet in memory
30              updateMinimumDistanceTileKey(camera, currentTileKey,
        minimumDistanceTileKey);
31              visibleNodes.push(node);
32              requestChildren(currentTileKey);
33          } else {
34              // Traverse the four children
35              collectRenderable(camera, currentTileKey.topLeftChild(),
        visibleTiles, minimumDistanceTileKey);
36              collectRenderable(camera, currentTileKey.topRightChild(),
        visibleTiles, minimumDistanceTileKey);
37              collectRenderable(camera, currentTileKey.bottomLeftChild(),
        visibleTiles, minimumDistanceTileKey);
38              collectRenderable(camera, currentTileKey.bottomRightChild(),
        visibleTiles, minimumDistanceTileKey);
39          }
40      }
41  }
```

Listing 4.17: Method `collectVisible()` in `TerrainManager` which collects all visible terrain nodes and requests unload ones.

The collected nodes are stored `visibleTiles`, which is a mutable reference passed by the caller `TerrainManager::render()`. The function call to `TerrainManager::updateMinimumDistanceTileKey()` is for the collision detection described later.

The function `TerrainManager::allChildrenExistant()` simply checks whether all child nodes are loaded in the memory cache, as shown in listing 4.18.

```
1   bool TerrainManager::allChildrenExistant(XYZTileKey tileKey)
2   {
3       return _memoryCache.contains(tileKey.topLeftChild())
4           && _memoryCache.contains(tileKey.topRightChild())
5           && _memoryCache.contains(tileKey.bottomLeftChild())
6           && _memoryCache.contains(tileKey.bottomRightChild());
7   }
```

Listing 4.18: Method `allChildrenExistant()` which checks whether all child nodes are loaded in memory.

The method `requestChildren()` requests the child nodes of a terrain node with the given XYZ tile key. A child node is only requested if it satisfies the following requirements:

- It is not in the memory cache already.
- It is not currently being loaded already.
- It is not unloadable (e.g. oceans at high zoom levels).
- It is not currently being deallocated from the disk cache.

These requirements ensure that no race conditions and unnecessary API requests occur. Listing 4.19 shows the pseudocode for requsting new nodes and checking for the above mentioned requirements.

```
1  void TerrainManager::requestChildren(XYZTileKey tileKey)
2  {
3      requestTile(tileKey.topLeftChild());
4      requestTile(tileKey.topRightChild());
5      requestTile(tileKey.bottomLeftChild());
6      requestTile(tileKey.bottomRightChild());
7  }
8
9  void TerrainManager::requestNode(XYZTileKey tileKey)
10 {
11     if (!_memoryCache.contains(tileKey) // Already loaded
12         && !_loadingNodes.count(tileKey) // Currently being loaded
13         && !_unloadableNodes.count(tileKey) // Cannot be loaded
14         && !_currentDiskCacheEvictions.count(tileKey)) { // Currently being
       evicted from disk cache
15         _loadingNodes.insert(tileKey);
16         LoadRequestType requestType = _diskCache.contains(tileKey) ?
       LOAD_REQUEST_DISK_CACHE : LOAD_REQUEST;
17         _loadRequestQueues[_currentLoadThread]->push({ tileKey, requestType,
       _offlineMode });
18         _currentLoadThread = (_currentLoadThread + 1) % _numLoadWorkers;
19     }
20 }
```

Listing 4.19: Methods `TerrainManager::requestChildren()` and `TerrainManager::requestNode()` for requesting new nodes.

After checking whether the node can be requested (lines 11 to 14), we add the tile key to the list of tile keys whose nodes are currently being loaded `_loadingNodes` (line 15) and create the load request (line 16). Based on whether the disk cache contains the node, we either send a request to load from the disk cache with `LOAD_REQUEST_DISK_CACHE` or with `LOAD_REQUEST` if we should load it from the web APIs. Afterwards, we put the request on the current load request queue (line 17), which is given by the integer `_currentLoadThread`. After that, we increment `_currentLoadThread` and if it exceeds the number of load workers, we set it to 0 again (line 18). In this manner, we cycle through

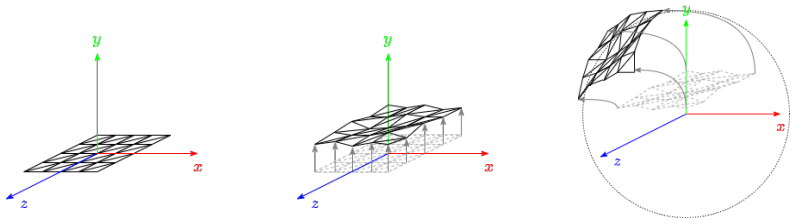
the request queues in a round-robin manner and ensure that the requests get distributed somewhat evenly over the load workers.

4.10.2 Rendering Visible Nodes

Now that all visible nodes have been collected in the render list, they need to be rendered. For each node, the uniforms of the terrain mesh shader and the skirt shader are updated and the overlay and heightmap textures bound. Afterwards, the terrain mesh and the skirt mesh get rendered, each with a single `glDrawElements()` call.

Vertex Shader

As already mentioned, the terrain mesh and the skirt mesh work almost identically. The vertex and fragment shader code for the terrain and the skirt mesh are essentially identical, except for a few small differences. In order to avoid redundant explanations, we will only outline the terrain mesh shader for the remainder of this subsection and explain the differences to the skirt shader where needed.



(a) (b) (c)

Figure 4.16: Illustration of what happens to the vertices of a terrain mesh in the terrain mesh vertex shader. Initially, the terrain mesh is lying flat at the null coordinate in subfigure (a). In the vertex shader, the height gets sampled from the heightmap texture and the y-coordinate of the vertex is displaced as shown in subfigure (b). Finally, the vertex is projected to the ellipsoid in subfigure (c).

The pseudocode of the `main()` function of the vertex shader for the terrain mesh is shown in listing 4.20.

```
1  void main()
2  {
3      float tw = tileWidth - 1;
4
5      // Normalized positions in range [0,1]
6      vec2 aPos1 = vec2((aPos.x + 0.5f * tw) / tw,
7                        (aPos.y + 0.5f * tw) / tw);
8
9      // Calculate Web mercator Coordinates
10     float mercX = (tileKey.x + aPos1.x) / float(1 << int(zoom));
11     float mercY = (tileKey.y + aPos1.y) / float(1 << int(zoom));
12
13     // Retrieve height form heightmap (important: denormalize to 255)
14     vec3 height = texture(heightmapTexture, aPos1).rgb * 255;
15
```

```
16      // Calculate height
17      float y = calculateHeight(height);
18
19      // Globe projection
20      vec2 lonlat = inverseWebMercator(vec2(mercX, mercY));
21      vec3 ellipsoidPos = geodeticToCartesian(globeRadiusSquared, vec3(lonlat.x
        , y, lonlat.y));
22
23      inTexCoords = aPos1;
24      inNormal = geodeticSurfaceNormal(vec3(lonlat.x, y, lonlat.y));
25      FragPosition = ellipsoidPos;
26      gl_Position = projection * view * model * vec4(ellipsoidPos, 1.0);
27  }
```

Listing 4.20: The `main()` function for the vertex shader.

For the skirt mesh, the height of a skirt vertex would additionally be subtracted by the skirt height after the retrieval of the y value (line 17).

The height calculation occurs with a special scaling in order to account for the reduced globe radii as shown in listing 4.21.

```
1   float calculateHeight(vec3 height) {
2     // Maptiler Terrain RGB formula
3     float y = -10000 + (((height.r * 256.0f * 256.0f * 0.1) + (height.g *
       256.0f * 0.1) + (height.b * 0.1)));
4     return (y / 20169.51); // Scaling down the Earth radius (globeRadii.x /
       6378159.09)
5   }
```

Listing 4.21: The GLSL `calculateHeight()` function for computing the height value from a RGB triple.

Fragment Shader

The fragment shader performs the texturing, applies an ambient light and the distance fog. The uniform variable `useWire` and `doFog` signals whether to render in wireframe mode and with fog respectively. The `main()` function of the fragment shader is shown in listing 4.22. The terrain mesh and the skirt mesh use essentially the same fragment shader code.

```
1   void main()
2   {
3       vec3 color;
4       vec3 lightColor = vec3(1.0f, 1.0f, 1.0f);
5
6      if (useWire < 0.5f) {
7          float tw = tileWidth - 1;
8          color = texture(overlayTexture, inTexCoords).rgb;
9          vec3 ambient = calculateAmbient(lightColor, 0.1f);
10
11         if (doFog > 0.5) {
12             vec3 fogColour = vec3(97, 154, 232) / 255.0f;
13             float fogFactor = calculateFog(fogDensity);
14             color = mix(fogColour, color, fogFactor);
15         }
16      } else {
17          color = terrainColor.rgb;
18      }
```

```
19      FragColor = vec4(color, 1.0f);
20  }
```

Listing 4.22: The `main()` method for the fragment shader.

The distance fog calculation is partially based on the the predecessor project [Tab24] and is shown in listing 4.23:

```
1  float calculateFog(float density) {
2      float start = 0.8;
3      float end = 3.5;
4      float dist = length(cameraPos - FragPosition);
5      if (dist < start) return 1.0f;
6      float scale = (dist - start) / (end - start);
7      float fogFactor = pow(scale, 1);
8      return 1.0f - clamp(fogFactor, 0.0f, 0.5f);
9  }
```

Listing 4.23: GLSL pseudocode for the distance fog.

4.11 Miscellaneous Features

This section describes some miscellaneous features implemented in StreamingATLOD.

4.11.1 Camera

The implemented camera is partially based on the chapter "Camera" of *Learn OpenGL* by de Vries [dV20], but is slightly modified for Earth rendering. Moving with the WASD-keys (W for forward, A for left, S for backward, D for right) moves the camera in the corresponding directions, but in such a way that the Earth stays underneath the camera at all times. With the Q and E keys, the altitude can be decreased and increased respectively. Dragging to the left and right with the mouse rotates the camera to the left and right, but so that the rotation occurs about the vector which goes from the closest point on Earth's surface to the camera's position. Dragging the mouse to the top and bottom rotates the camera up and down, with the pitch angle being limited to the range of $-89°$ to $0°$, so that the camera cannot point directly down towards the ground and up towards the sky. Figure **??** shows an illustration of the Earth-centered camera movement.

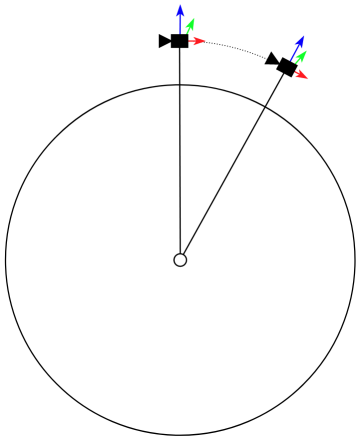


Figure 4.17: Illustration of the Earth-centered camera. The front vector is red, the right vector green and the up vector blue.

Upon keyboard input by the user, we perform the following: If the camera should move to the left or right, we simply add the right vector to the current position and we are done. Otherwise, if the camera should move up, down, forward or backwards, we first calculate the Earth's normal vector at the position of the camera by taking the length of the camera's position vector, since the Earth is centered at $(0, 0, 0)$. If the camera should move up and down, we multiply the Earth normal vector by the velocity and add it to the current position. If the camera should move forward or backward, we compute a new front vector by projecting the current front vector onto the plane orthogonal to the Earth normal. Afterwards, we multiply the projected front vector by the camera velocity and add the new projected front vector to the current position vector. In addition to the above steps, we limit the camera's position to always stay inside a sphere of radius 1000, so that the camera doesn't fly away.

When updating the camera vectors, we first create a quaternion `earthQuat` where the up vector points towards the Earth normal vector. Afterwards, we first rotate the unit front vector according to the Earth with `earthQuat`, followed by the yaw and pitch rotations. Afterwards, we normalize the vectors and compute the right and up vectors as well. Listing 4.24 shows the updating of the camera vectors.

```
1  void Camera::updateCameraVectors()
2  {
3      glm::vec3 earthNormal = glm::normalize(_position);
4
5      glm::vec3 earthRight = glm::normalize(glm::cross(earthNormal, _worldUp));
6      glm::vec3 earthFront = -1.0f * glm::normalize(glm::cross(earthNormal,
       earthRight));
7      glm::vec3 front1;
8
9      front1 = glm::vec3(0, 0, -1.0);
10
11     glm::mat4 earthMat = glm::mat4(glm::vec4(earthRight, 0),
12         glm::vec4(earthNormal, 0),
13         glm::vec4(earthFront, 0),
14         glm::vec4(0, 0, 0, 1));
15
```

```
16      glm::quat pitchQuat = glm::angleAxis(glm::radians(_pitch), earthRight);
17      glm::quat yawQuat = glm::angleAxis(glm::radians(_yaw), earthNormal);
18      glm::quat earthQuat = glm::quat_cast(earthMat);
19
20      front1 = glm::vec3(yawQuat * pitchQuat * earthQuat * front1);
21
22      _front = glm::normalize(front1);
23      _right = glm::normalize(glm::cross(_front, earthNormal));
24      _up = glm::normalize(glm::cross(_right, _front));
25  }
```

Listing 4.24: Method `Camera::updateCameraVectors()` that updates the camera vectors.

4.11.2 Collision Detection

The collision detection prevents the camera from flying beneath the Earth. It is implemented using a very simple heightmap check.

First, we need to determine which terrain node's heightmap to use for collision detection. We use the following simple observation: At any point during rendering, the camera is always above a single leaf terrain node.

During the collection traversal, when we reach a leaf terrain node which should be rendered, we check whether the camera's longitude and latitude is contained inside the terrain node's longitudal and latitudal extents. If so, we track the node's tile key so that the terrain node can later be used for collision detection. For checking whether the camera's longitude and latitude is contained inside the node's longitude and latitude bounding rectangle, the camera's position is first transformed to the Web Mercator projection using `MapProjections::webMercator()`. The terrain tile's corner points are also projected to Web Mercator using the same method. This updating of the closest tile key is encapsulated in the method `TerrainManager::updateMinimumDistanceTileKey()` shown in listing 4.25.

```
1   void TerrainManager::updateMinimumDistanceTileKey(Camera& camera,
      XYZTileKey currentTileKey, XYZTileKey& minimumDistanceTileKey)
2   {
3       glm::vec2 cameraLonLat = MapProjections::toGeodetic2D(camera.position()
      , GlobalConstants::GLOBE\_RADII\_SQUARED);
4       glm::vec2 cameraMerc = MapProjections::webMercator(cameraLonLat);
5
6       float pow2Zoom = 1 << currentTileKey.z();
7       float minMercX = (currentTileKey.x()) / pow2Zoom;
8       float maxMercX = (currentTileKey.x() + 1) / pow2Zoom;
9       float minMercY = (currentTileKey.y()) / pow2Zoom;
10      float maxMercY = (currentTileKey.y() + 1) / pow2Zoom;
11
12      // If camera is inside leaf node, update tile key
13      if (cameraMerc.x >= minMercX && cameraMerc.x <= maxMercX && cameraMerc.
      y >= minMercY && cameraMerc.y <= maxMercY)
14          minimumDistanceTileKey = currentTileKey;
15  }
```

Listing 4.25: Method `TerrainManager::updateMinimumDistanceTileKey()` that updates the tile key of the minimum distance terrain node.

After the collection traversal, we compute the local heightmap coordinate that we need to retrieve for checking the height value with the camera's height value and perform the actual collision detection. This is implemented in `TerrainManager::checkCollision()` as shown in listing 4.26.

```
1  void TerrainManager::checkCollision(Camera& camera, XYZTileKey
       minimumDistanceTileKey)
2  {
3      TerrainNode* node = _memoryCache.get(minimumDistanceTileKey).value;
4      glm::vec2 lonLat = MapProjections::toGeodetic2D(camera.position(),
        GlobalConstants::GLOBE_RADII_SQUARED);
5      glm::vec2 camMerc = MapProjections::webMercator(lonLat);
6
7      float minDistZoomPow2 = 1 << minimumDistanceTileKey.z();
8
9      // Compute minimum and maximum Web Mercator coordinates.
10     float minMercX = (minimumDistanceTileKey.x()) / minDistZoomPow2;
11     float maxMercX = (minimumDistanceTileKey.x() + 1) / minDistZoomPow2;
12     float minMercY = (minimumDistanceTileKey.y()) / minDistZoomPow2;
13     float maxMercY = (minimumDistanceTileKey.y() + 1) / minDistZoomPow2;
14
15     float heightX = ((camMerc.x - minMercX) / (maxMercX - minMercX));
16     float heightY = ((camMerc.y - minMercY) / (maxMercY - minMercY));
17     glm::vec2 lonLatTerrain = MapProjections::inverseWebMercator(glm::vec2(
        heightX, heightY));
18
19     verticalCollisionOffset = 0.0;
20
21     // Check if inside bounds
22     if (heightX >= 0 && heightX <= 1 && heightY >= 0 && heightY <= 1) {
23         // Retrieve height value
24         float height = node->getScaledHeight(heightX * 511, heightY * 511);
25
26         // Project height point onto ellipsoid
27         glm::vec3 projected = MapProjections::geodeticToCartesian(
        GlobalConstants::GLOBE_RADII_SQUARED, glm::vec3(lonLatTerrain.x, height
        + 0.04, lonLatTerrain.y));
28
29         float distCamera = glm::length(camera.position());
30         float distHeight = glm::length(projected);
31
32         // Collision
33         if (distHeight > distCamera) {
34             camera.verticalCollisionOffset(distHeight - distCamera);
35         }
36     }
37 }
```

Listing 4.26: Method `TerrainManager::checkCollision()` that performs the collision detection.

First, we transform the camera's world-space coordinates into Web Mercator (lines 4 and 5). Afterwards, compute the bounding rectangle of the closest node whose tile key is `minimumDistanceTileKey` (lines 10 to 13). Next, we compute the heightmap coordinates `heightX` and `heightY` by normalizing the camera's Web Mercator coordinates (lines 15 and 16) and subsequently compute the 2D geodetic coordinates `lonLatTerrain` of the point on the terrain to test (line 17). We then check whether the `heightX` and `heightY` are inside the range $[0, 1]$ (line 22) and if so, retrieve the height value by scaling `heightX` and `heightY` by the image width and height respectively and calling `TerrainNode::getScaledHeight()` (line 24). With the newly retrieved height value plus a small offset value and `lonLatTerrain`, we compute the actual terrain point

in cartesian coordinates to compare with the camera position (line 27). Since the Earth is centered around $(0, 0, 0)$, if suffices to compare the length of both vectors (lines 29 and 30). If the position of the terrain point is higher than the camera's position, we set the vertical collision offset value in the camera (lines 33 and 34) such that it can later correct the offset and thus not fly under the Earth.

4.11.3 Skybox

Currently, StreamingATLOD uses a simple cubemap for rendering the sky. The implementation of the cubemap is based on chapter "Cubemaps" from *Learn OpenGL* by de Vries [dV20]. At runtime, the skybox is rotated such that it always points in the direction of the normal vector of the Earth relative to the camera, such that the blue sky always points upwards.

4.11.4 Application Configuration

StreamingATLOD can be configured using a configuration file named `config.txt`. The configuration file consists of simple key-value pairs per line, with the key and value delimited by an equal symbol (=). The following options must be set:

- Heightmap service URL: The URL for the heightmap web API.
- Overlay service URL: The URL for the overlay web API.
- Heightmap service key: The key for the heightmap web API.
- Overlay service key: The key for the overlay web API.
- Maximum zoom level: The maximum zoom level a terrain node can reach. Limited to between 0 and 30. Setting this value higher than what the APIs can serve risks making unnecessary API requests.
- Memory cache size: The maximum number of elements in the memory cache. Limited to between 100 and 500.
- Disk cache size: The maximum number of elements in the disk cache. Limited to between 400 and 8000. Also, the disk cache capacity must be at least four times the memory cache capacity.
- Disk cache location: The location of the disk cache on the file system.
- Number of load workers: The number of load worker threads. Limited to between 1 and 8.
- Low resolution mesh size: The side length of the low resolution terrain mesh. Limited to between 8 and 512.
- Medium resolution mesh size: The side length of the medium resolution terrain mesh. Limited to between 8 and 512.
- High resolution mesh size: The side length of the high resolution terrain mesh. Limited to between 8 and 512.

- Data folder location: The location of the data folder, which contains the GLSL shader code and skybox images.

The configuration file is loaded and validated at startup and is managed as a singleton class `ConfigManager`. After the loading of the configuration file, the configuration settings stay constant throughout the runtime of StreamingATLOD and are not changed.

The details on how to set configurations and launch StreamingATLOD with them are described in the `README` of StreamingATLOD's GitHub repository[1].

4.11.5 Debug Visualizations

Various minor debug visualizations were implemented in order to showcase how the various implemented techniques work.

Camera Freezing

Freezing the camera means that the logical camera (i.e. the position and front vectors and the view-frustum) get frozen in place, while the actual camera can still be flown around. With a frozen camera, the culling techniques and the LOD calculation get forzen in place as well. For this, the application always keeps track of two varaibles: `Camera oldCamera` and `Camera camera`. Every frame, `oldCamera` is set to `camera` and subsequently `oldCamera` is passed to the `TerrainManager::render()` method. If the user activates camera freezing, then `oldCamera` is no longer updated to be `camera`. This means that `TerrainManager::render()` continues to compute culling and LOD with `oldCamera`, while `camera` is used for the camera movement and the model-view-projection matrix. Figure 4.18 shows an example of camera freezing.

(a) (b)

Figure 4.18: The view from the frozen camera shown in subfigure (a) what actually gets rendered as from the frozen camera in subfigre (b).

[1] `https://github.com/amartabakovic/bachelor-thesis`

Rendering AABBs

For rendering AABBs, there is an `AABBMesh` class and a corresponding shader consisting of `aabbmesh.vert` and `aabbmesh.frag`. The AABB mesh, like the terrain, skirt and pole meshes, is defined once during startup and stored as a member in `TerrainManager`. During rendering, the AABB is scaled and translated using the terrain node's minimum and maximum points.

An example of an AABB render is shown in figure 4.19.

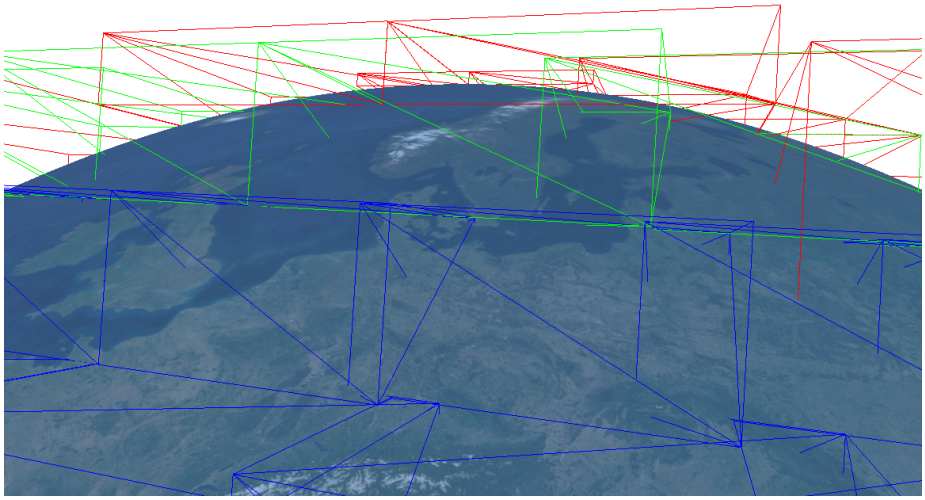


Figure 4.19: Screenshot of a debug AABB render.

Wireframe Mode

The wireframe of the terrain is rendered using the OpenGL command `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`. Prior to the draw call, the uniform `vec3` variable `terrainColor` of the terrain shader is set to either red, green or blue, according to the following listing:

```
1  // Set colors for debug wireframe view
2  glm::vec3 terrainColor = glm::vec3(1, 0, 0);
3  if (wireframe) {
4      if (zoom % 3 == 1)
5          _terrainShader.setVec3("terrainColor", glm::vec3(0, 1, 0));
6      else if (zoom % 3 == 2)
7          _terrainShader.setVec3("terrainColor", glm::vec3(0, 0, 1));
8  }
9
10 _terrainShader.setVec3("terrainColor", terrainColor);
```

Listing 4.27: Setting the color of the terrain for the wireframe mode.

An example of a wireframe render is shown in figure 4.20.

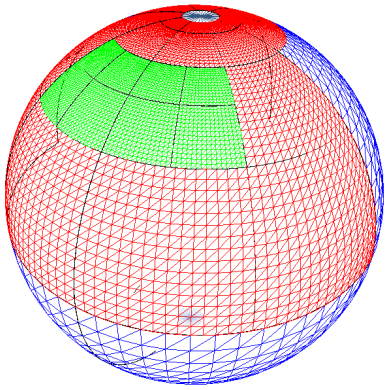


Figure 4.20: Screenshot of a debug wireframe render.

Chapter 5

Results

This chapter gives the results of various benchmarks performed on StreamingATLOD. Two types of benchmarks are performed. The first type are 5 flight experiments, whose main goal is to benchmark how much time StreamingATLOD needs to stream in new data when the disk cache is empty and how well StreamingATLOD performs in terms of rendering performance. A single flying experiment consists of a simple flight starting from outer space towards a destination on the Earth and aims to show the following:

- A screenshot of the destination place at the highest available resolution.
- The number of requests StreamingATLOD performs when flying to a place for the first time and the amount of time it requires until loading all required data, including the time between arriving at the destination and receiving the last API response.
- The average framerate of such a flight.
- The number of traversed terrain nodes, draw calls and rendered triangles when vieweing the destination place (at the state of the screenshot).
- The total memory consumption done by StreamingATLOD when viewing the destination place.

As such, all flight experiments were conducted with the disk cache empty upon every start up. The time measurements were performed manually with a stopwatch app on a mobile phone since the time differences are in the range of multiple seconds. The total memory consumption (CPU + GPU) is retrieved from the "Activity Monitor" app on macOS.

The second type of benchmarking performed is the measurement of the disk cache size for various disk cache capacities. Since the overlay and heightmap tiles are encoded in JPG and WebP on the disk respectively, not all tiles use up the same amount of memory.

5.1 Experimental Setup

5.1.1 Used Hardware

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

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

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

5.1.2 Application Configuration

The application uses the following configuration settings:

- Low resolution mesh size: 8
- Medium resolution mesh size: 32
- High resolution mesh size: 32
- Number of load worker threads: 6
- Heightmap data set: MapTiler RGB
- Overlay data set: MapTiler Satellite
- Memory cache size: 400 nodes
- Disk cache size: for the flight experiments 8000 nodes, for the disk cache measurements 400, 2000 and 8000 nodes

The application window resolution is set to 1280×720.

5.2 Flight Experiments

5.2.1 Experiment 1: The Swiss Alps

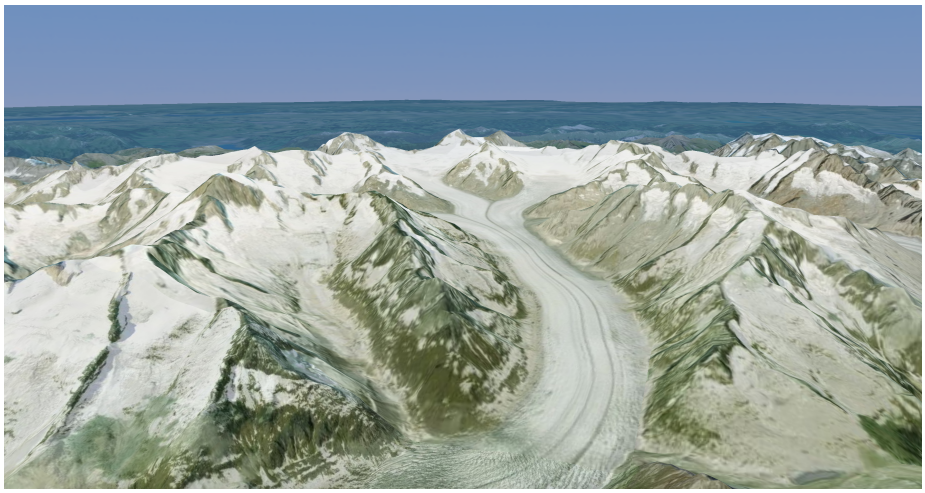


Figure 5.1: Screenshot of experiment 1: The Swiss Alps.

Time until destination reached t_{dest}	29 seconds
Time until all tiles loaded t_{load}	38 seconds
$t_{load} - t_{dest}$	9 seconds
Draw calls	126
Rendered triangles	86002
Visible nodes	62
Traversed nodes	233
Highest zoom level	13
Nodes in memory cache	233
Average FPS	64.2
API requests	466
GPU memory for textures	426.9 MB
Total memory (CPU + GPU) consumption	901.5 MB

Table 5.2: Benchmarks for experiment 1.

5.2.2 Experiment 2: The Grand Canyon

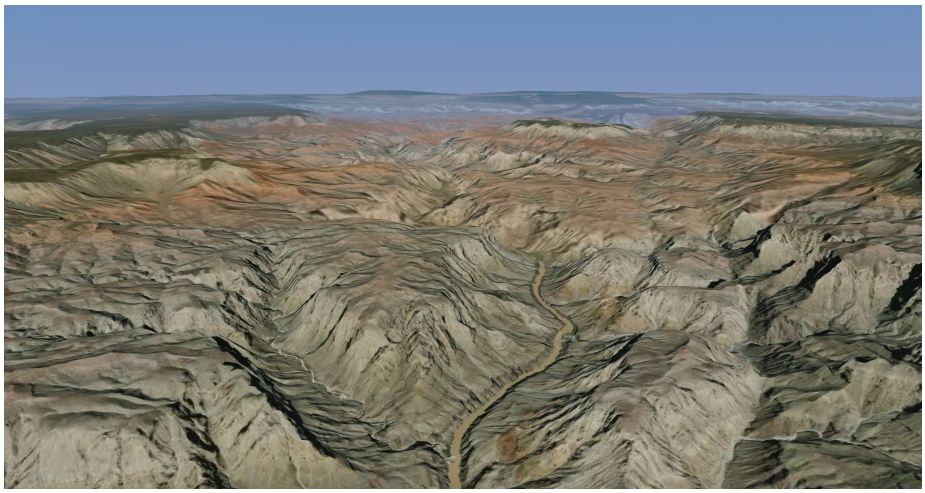


Figure 5.2: Screenshot of experiment 2: The Grand Canyon.

Time until destination reached t_{dest}	35 seconds
Time until all tiles loaded t_{load}	45 seconds
$t_{load} - t_{dest}$	10 seconds
Draw calls	118
Rendered triangles	75422
Visible nodes	58
Traversed nodes	107
Highest zoom level	13
Nodes in memory cache	217
Average FPS	71.2
API requests	490
GPU memory for textures	397.6 MB
Total memory (CPU + GPU) consumption	824.4 MB

Table 5.3: Benchmarks for experiment 2.

5.2.3 Experiment 3: Northern Iceland

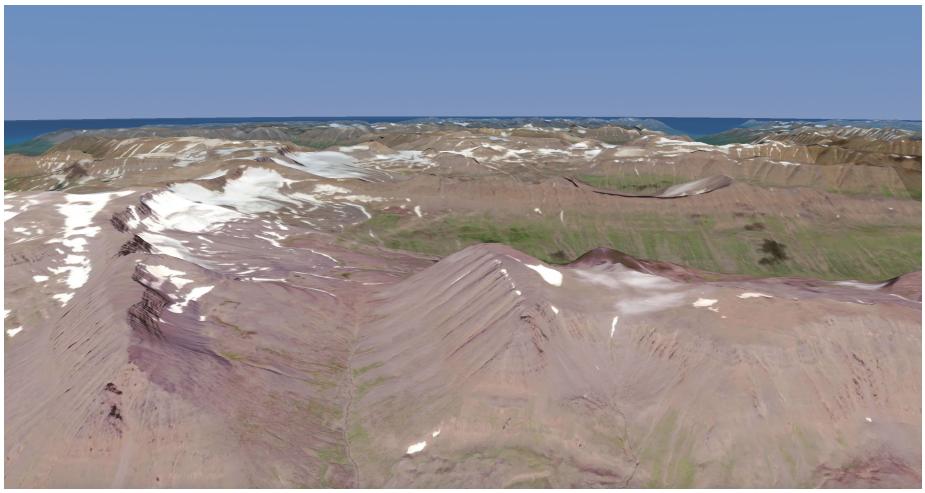


Figure 5.3: Screenshot of experiment 3: Northern Iceland.

Time until destination reached t_{dest}	35 seconds
Time until all tiles loaded t_{load}	66 seconds
$t_{load} - t_{dest}$	31 seconds
Draw calls	214
Rendered triangles	151070
Visible nodes	106
Traversed nodes	182
Highest zoom level	14
Nodes in memory cache	309
Average FPS	56.4
API requests	628
GPU memory for textures	566.2 MB
Total memory (CPU + GPU) consumption	1.13 GB

Table 5.4: Benchmarks for experiment 3.

5.2.4 Experiment 4: The Andes Mountains

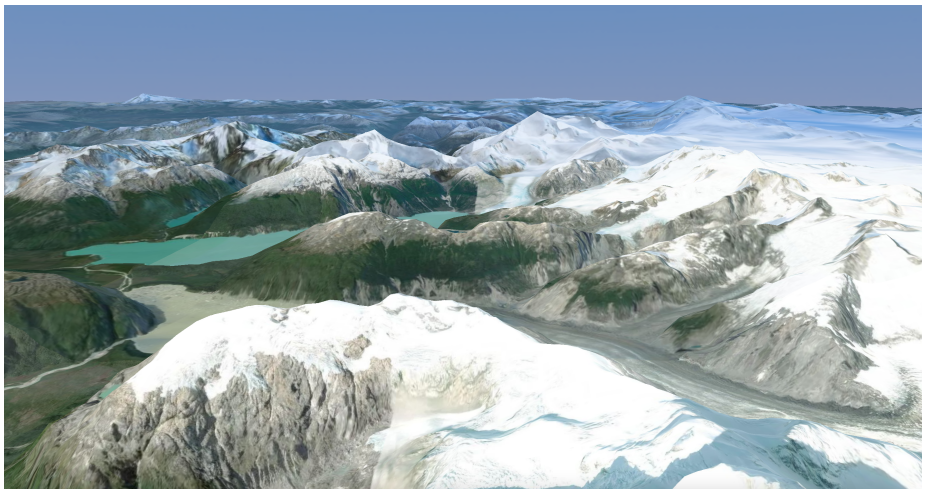


Figure 5.4: Screenshot of experiment 4: The Andes mountains.

Time until destination reached t_{dest}	30 seconds
Time until all tiles loaded t_{load}	55 seconds
$t_{load} - t_{dest}$	25 seconds
Draw calls	138
Rendered triangles	85150
Visible nodes	68
Traversed nodes	119
Highest zoom level	14
Nodes in memory cache	278
Average FPS	66.9
API requests	626
GPU memory for textures	509.4 MB
Total memory (CPU + GPU) consumption	1.04 GB

Table 5.5: Benchmarks for experiment 4.

5.2.5 Experiment 5: The Himalayas

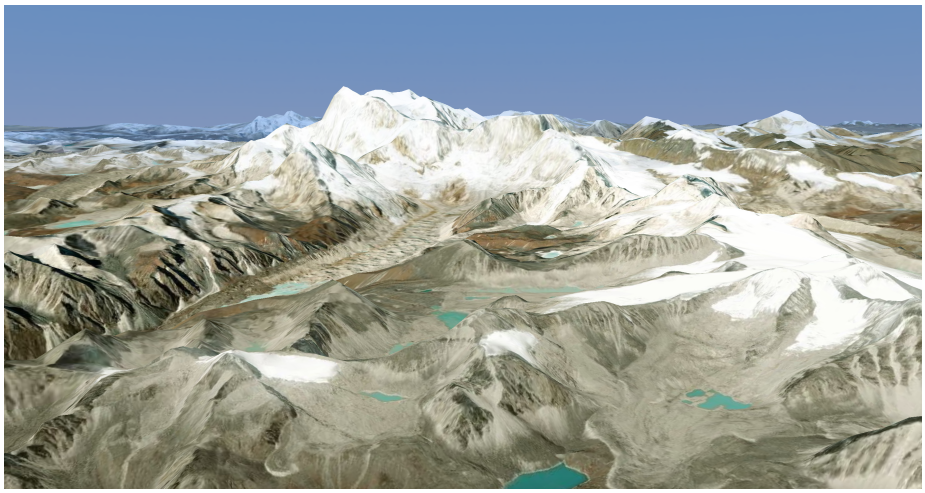


Figure 5.5: Screenshot of experiment 5: The Himalayas.

Time until destination reached t_{dest}	30 seconds
Time until all tiles loaded t_{load}	46 seconds
$t_{load} - t_{dest}$	16 seconds
Draw calls	116
Rendered triangles	78110
Visible nodes	57
Traversed nodes	99
Highest zoom level	14
Nodes in memory cache	217
Average FPS	64.5
API requests	434
GPU memory for textures	397.6 MB
Total memory (CPU + GPU) consumption	821.4 MB

Table 5.6: Benchmarks for experiment 5.

5.2.6 Disk Cache Size Measurements

Disk cache capacity	Heightmap subfolder	Overlay subfolder	total
400	74 MB	18 MB	92 MB
2000	311 MB	93 MB	404 MB
8000	1117 MB	349 MB	1466 MB = 1.4 GB

Table 5.7: Disk cache size measurements for the capacities 400, 2000 and 8000.

Chapter 6

Discussion

This chapter discusses the results obtained in the previous chapter.

6.1 Rendering Performance

In terms of rendering performance, StreamingATLOD performs decently well given the hardware it was tested on. With the used configuration settings, the triangle count is kept well below 200000 and the number of draw calls below 300 even at the highest zoom levels, without sacrificing the visual appearance too much. The framerates are approriate as well in most cases, with only a few occasional hiccups while close to the ground.

6.2 Streaming Performance

The average time between arriving at a place and finishing loading the necessary tiles were on average 18.2 seconds. Given the fact that the disk cache was empty in all experiments, some latency was to be expected. As such, the main bottleneck of the streaming performance lies in the requesting from the web APIs. Disk caching prevents this temporarily, especially for places which are visited frequently by the user.

6.3 Memory Usage

In the experiments, the total memory usage ranged between 800 MB to 1.2 GB, which is a normal range for a large-scale Earth renderer. Upon reaching the memory cache capacity size, the memory allocated for terrain data remains practically constant, since we always evict unused terrain nodes upon loading new ones in. Once the disk cache is full, then the memory consumption in total stays constant, since we do not have to store new tile keys in the in-memory disk cache anymore.

6.4 Disk Cache Measurements

The file system size of the disk cache is kept mostly in a normal range, which is between 92 MB for a disk cache capacity of 400 nodes and 1.4 GB for a disk cache capacity of 8000 nodes. The heightmap subfolder of the file system disk cache is in most cases around three to four times larger than the overlay subfolder. The reason for this is that the heightmap tiles are stored in WebP format instead of JPG.

Chapter 7

Conclusion

7.1 Potential Improvements

This section lists some potential technical improvements to the rendering and the streaming aspects of StreamingATLOD.

Higher Precision Support Currently, StreamingATLOD operates on 32-bit floating point numbers for most purposes, including vertex transformations. So far, this has not been a problem, since the terrain data was limited to zoom level 14, which corresponds to a maximum precision of 9.5 meters per pixel. However, if StreamingATLOD were to be used with centimeter-accurate terrain data, this would likely result in jittering artifacts and depth buffer fighting during rendering, requiring modifications described in section in section "Potential Problems during Rendering" of chapter "Theoretical Background".

Request Priorization When new tiles must be loaded into memory, StreamingATLOD pushes these requests into a request queue. Currently, a worker thread processes these requests in the order which they came in, meaning from oldest to newest, which is not ideal, since old requests which might not be needed anymore still have to get loaded. An improvement would be to prioritize requests.

Transitions Between LOD Changes Right now, when a terrain node with zoom level gets rendered and at some point its four children must get rendered instead, the sudden change between the lower resolution and the higher resolution nodes causes a visible popping artifact of higher detailed terrain to occur. One way to reduce popping is to quickly blend the terrain meshes together, thus reducing the popping. This comes at the cost of having to render both the high resolution and low resolution meshes at the same time for a short while.

7.2 Possibilities for Extension

This section describes some potential features and ideas on how StreamingATLOD could be further enhanced.

Extension into a Software Development Kit StreamingATLOD could be extended into a software development kit (SDK) for developing desktop-based Earth renderers with C++ and OpenGL. For this, various classes would have to be restructured for more flexibility and configurability.

Multiple Data Layers Currently, StreamingATLOD operates on one height data layer and one imagery layer. StreamingATLOD could be extended to be able to work with multiple data layers. This would be especially useful in case data is not available in one layer but might be available in another.

Realistic Atmospheric Rendering StreamingATLOD currently only renders the black color if the camera is in space and a static skybox for when the camera is close to the ground and interpolates between both if the camera is in between. A realistic atmospheric model, such as *Precomputed Atmospheric Scattering* by Bruneton and Neyret [BN08] would increase the realism and the visual aesthetics.

7.3 Personal Conclusion

This conclusion is my personal reflexion and outlook on this bachelor thesis. Generally, I am very satisfied with the system I've developed. This was probably the most challenging project I've worked on. I am very happy with the results and was able to finish almost everything I set out to do.

Bibliography

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

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

[con24] PROJ contributors. Web mecator / pseudo mercator. `https://proj.org/en/9.4/operations/projections/webmerc.html`, 2024.

[CR11] Patrick Cozzi and Kevin Ring. *3D Engine Design for Virtual Globes*. A. K. Peters, Ltd., USA, 1st edition, 2011.

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

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

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

[ec] everything curl. Connection reuse. `https://everything.curl.dev/transfers/conn/reuse.html`.

[Fue22] Lionel Fuentes. Advanced graphics summit: Designing the terrain system of 'flight simulator': Representing the earth. `https://www.gdcvault.com/play/1027581/Advanced-Graphics-Summit-Designing-the`, 2022.

[Inc] Apple Inc. Opengl programming guide for mac. `https://developer.apple.com/library/archive/documentation/GraphicsImaging/Conceptual/OpenGL-MacProgGuide/opengl_threading/opengl_threading.html`.

[LCN12] Benjamin Loesch, Martin Christen, and Stephan Nebiker. Openwebglobe - an open source sdk for creating large-scale virtual globes on a webgl basis. *ISPRS - International Archives of the Photogrammetry,*

Remote Sensing and Spatial Information Sciences, XXXIX-B4:195–200, 07 2012.

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

[LP01] Peter Lindstrom and Valerio Pascucci. Visualization of large terrains made easy. 10 2001.

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

[Map] MapTiler. Tiles à la google maps coordinates, tile bounds and projection. `https://docs.maptiler.com/google-maps-coordinates-tile-bounds-projection/`.

[Map23] MapTiler. Rgb terrain by maptiler. `https://documentation.maptiler.com/hc/en-us/articles/4405444055313-RGB-Terrain-by-MapTiler`, 2023.

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

[Pet12] Michael Peterson. *The Tile-Based Mapping Transition in Cartography*, pages 151–163. 01 2012.

[Red] Redis. What is lru cache? `https://redis.io/glossary/lru-cache/`.

[Rin13] Kevin Ring. Horizon culling. `https://cesium.com/blog/2013/04/25/horizon-culling/`, 2013.

[Tab24] Amar Tabakovic. 3d terrain with level of detail, 2024. Report for semester project "Project 2".

[UCfGIS] GIS&T Body of Knowledge University Consortium for Geographic Information Science. Dm-10 - triangular irregular network (tin) models. `https://gistbok.ucgis.org/bok-topics/triangular-irregular-network-tin-models`.

[Ulr02] Thatcher Ulrich. Rendering massive terrains using chunked level of detail control. *SIGGRAPH 2002 Super-Size It! Scaling Up to Massive Virtual Worlds Course Notes*, 2002.

[Whi] Timothy Whitehead. How big is the google earth database? `https://www.gearthblog.com/blog/archives/2016/04/big-google-earth-database.html`.

[Wik24a] Wikipedia contributors. Popping (computer graphics) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Popping_(computer_graphics)&oldid=1228037078`, 2024. [Online; accessed 9-June-2024].

[Wik24b] Wikipedia contributors. Tiled web map — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Tiled_web_map&oldid=1222234446, 2024. [Online; accessed 6-June-2024].

[Wik24c] Wikipedia contributors. Tissot's indicatrix — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Tissot%27s_indicatrix&oldid=1218991137, 2024. [Online; accessed 9-June-2024].

[Wik24d] Wikipedia contributors. Web mercator projection — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Web_Mercator_projection&oldid=1228775025, 2024. [Online; accessed 13-June-2024].

Appendix A

Time Schedule

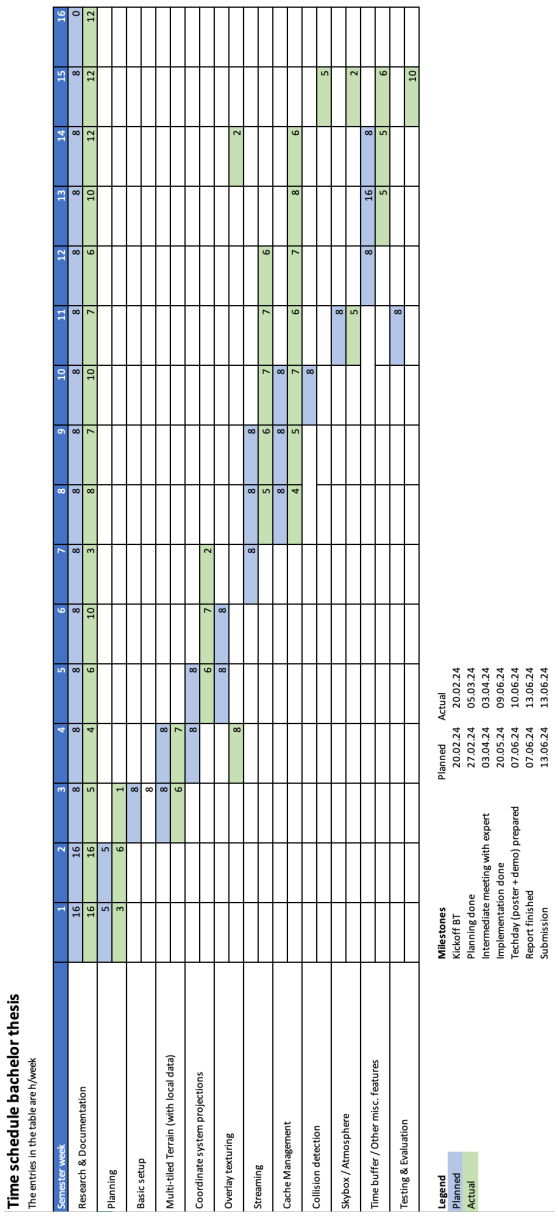


Figure A.1: The time schedule of this thesis.