UNIVERSITY OF TECHNOLOGY
IN THE EUROPEAN CAPITAL OF CULTURE
CHEMNITZ

Methods for Automated Creation and Efficient Visualisation of Large-Scale Terrains based on Real Height-Map Data

Kurt Kühnert

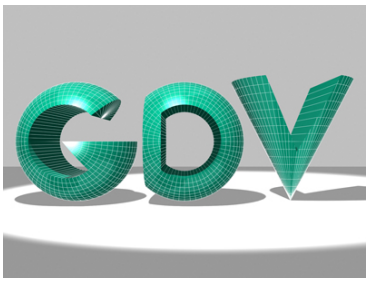
November 2022

A thesis submitted to the University of Technology Chemnitz in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

Kurt Kühnert: *Methods for Automated Creation and Efficient Visualisation of Large-Scale Terrains based on Real Height-Map Data* (2022)

Contact: kurt@kuehnert.dev

The work in this thesis was carried out in the:



Professorship of Computer Graphics and Visualization
Chemnitz University of Technology

Supervisors: Prof. Dr. Guido Brunnett
M. Sc. Tom Uhlmann

Abstract

Real-time rendering of large-scale terrains is a difficult problem and remains an active field of research. The massive scale of these landscapes, where the ratio between the size of the terrain and its resolution is spanning multiple orders of magnitude, requires an efficient level of detail strategy. It is crucial that the geometry, as well as the terrain data, are represented seamlessly at varying distances while maintaining a constant visual quality. This thesis investigates common techniques and previous solutions to problems associated with the rendering of height field terrains and discusses their benefits and drawbacks. Subsequently, two solutions to the stated problems are presented, which build and expand upon the state-of-the-art rendering methods. A seamless and efficient mesh representation is achieved by the novel **Uniform Distance-Dependent Level of Detail (UDLOD)** triangulation method. This fully GPU-based algorithm subdivides a quadtree covering the terrain into small tiles, which can be culled in parallel, and are morphed seamlessly in the vertex shader, resulting in a densely and temporally consistent triangulated mesh. The proposed **Chunked Clipmap** combines the strengths of both quadtrees and clipmaps to enable efficient out-of-core paging of terrain data. This data structure allows for constant time view-dependent access, graceful degradation if data is unavailable, and supports trilinear and anisotropic filtering. Together these, otherwise independent, techniques enable the rendering of large-scale real-world terrains, which is demonstrated on a dataset encompassing the entire Free State of Saxony at a resolution of one meter, in real-time.

Keywords: Terrain Rendering, UDLOD, Chunked Clipmap, Level of Detail, Height Field, GPU

Acknowledgments

I would like to express my gratitude to my primary supervisor, Prof. Dr. Guido Brunnett, for accepting this ambitious topic. Also, I greatly appreciate the helpful and responsive communication I have had with my secondary supervisor, Tom Uhlmann, who guided me through this project.

I am also grateful to my friend and fellow student Richard Wolf, who wrote his own bachelor thesis simultaneously, for our countless exchanges of ideas, for your help, and for your feedback on numerous revisions of this project.

Very special thanks to the Bevy community, for your friendly and open nature, to the countless people contributing to this project voluntarily, and to the ones who motivated and shared their knowledge with me. I would not have learned as much over the last year without you. The plugin developed as part of this thesis is dedicated to you.

Thanks to Nicola Papale and robtfm for reviewing my thesis and giving me valuable feedback.

I would also like to thank my friends and family who supported and motivated me. Special thanks to my parents for funding me during my studies.

Immense gratitude to Nathalie, my fiancée, for your patience and support.

Contents

Chapter 1

Introduction

Large-scale three-dimensional terrain rendering is becoming an increasingly popular and important feature in modern game, visualization, and graphic engines. Many applications, like racing/flight simulations or open-world games, are heavily reliant on this technique, for an immersive and realistic experience. Not only does the entertainment industry incorporate terrain rendering into their products but also geographical information systems (GIS), as well as city or construction planning tools, are using it. With the ever-increasing performance of consumer devices, it is becoming more and more common to integrate a real-time 3D representation of real-world landscapes into applications. This trend is becoming especially interesting with the possibility of displaying terrains on the web and in virtual reality, which will open up a multitude of new use cases, in the near future.

Because of these different fields of application, there has been immense interest in improving real-time terrain rendering since the advent of computer graphics. That is why many different methods of handling and visualizing those giant datasets have been developed over the last decades.

1.1 Problem Statement

Large-scale terrains are commonly represented as large two-dimensional textures, where each pixel stores the terrain's properties at the corresponding position. These properties may include the terrain's height and normal, but albedo or texturing information is also required by specific applications.

The brute force rendering method of assigning a vertex to each of the texels quickly surpasses the rendering capabilities of even modern hardware. Additionally, depending on the size and resolution of the terrain those textures usually exceed the capacity of random access memory (RAM) and video random access memory (VRAM), which is why an efficient out-of-core (OOC) level of detail (LOD) system is crucial for real-time terrain rendering.

This introduces two problems, which need to be solved.

Terrain Geometry Firstly, it is important to find a view-dependent mesh approximation of the loaded data that strikes a good balance between the triangle count, the quality (screen-space error), and the effort to compute and maintain this tessellation.

Terrain Data Secondly, the terrain data has to be streamed and managed, adaptive to the viewpoint. It is especially important that this data represents the terrain at any distance equally well and that it can be used for seamless high-quality texturing.

In recent years many solutions to these two problems have been developed, some of which tackle both issues in tandem while others are focusing on only one of those concerns.

1.2 Goal of this Thesis

In this bachelor thesis, the author will explore and give a brief overview of the terminology and general ideas of the problem space of large-scale real-time terrain rendering. The reader is expected to be familiar with the basics of computer graphics and the modern graphics pipeline. This is followed by a survey of a selection of different existing methods used to render height field terrain in real-time. Afterward, the author describes and presents two novel solutions to the aforementioned problems, as well as details about their implementation and their performance. A seamless and efficient mesh representation is achieved by the novel **Uniform Distance-Dependent Level of Detail (UDLOD)** triangulation algorithm. Additionally, the data structure named **Chunked Clipmap** is responsible for adaptively streaming and providing the terrain data inside the application.

This thesis is limited to terrain rendering using rasterized height fields only, although other techniques exist, such as ray casting, cube marching, or voxel rendering.

Chapter 2

Terminology

Before the existing terrain rendering algorithms can be examined, some common terminology has to be established. Both the **triangulation** and the **level of detail (LOD)** are crucial for approximating the mesh geometry, although they can not be strictly separated from each other, they can be classified into different categories. In addition, **visibility culling** plays an important role in optimizing the rendering of said geometry and will be examined in Section 2.3. Furthermore, **hardware tesselation**, and **terrain data** management are important prerequisites for surveying specific terrain rendering methods. Finally, a brief overview of the origin of **real-world terrain data** is presented in this chapter.

2.1 Triangulation

The terrain's surface must be represented by triangles, because they are the fundamental primitive of meshes in graphics application programming interfaces (APIs). The three most common methods for dividing a plane into triangles are **regular grids**, **triangulated irregular networks (TINs)** and **right-triangulated irregular networks (RTINs)** [1], as seen in Figure 2.1.

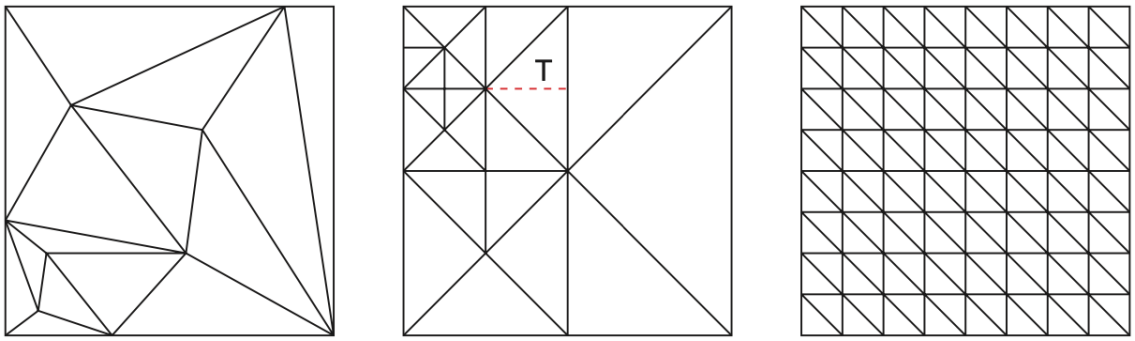


Figure 2.1: A comparison between a TIN on the left, a RTIN in the middle, and a regular grid on the right. (taken from [1])

2.1.1 Regular Grids

Regular grid subdivision [2] is arguably the simplest method of terrain triangulation. It subdivides the plane into a grid consisting of multiple uniform squares, which are then in turn split into two triangles. Subsequently, each vertex is then displaced by the terrain's height value. This step can either be accomplished in a preprocessing step, or it can be performed at runtime in the vertex shader, by fetching the value from the height map of the terrain.

2.1.2 Triangulated Irregular Networks

Contrary to regular grids - triangulated irregular networks (TINs) [2] distribute their vertices inside the plane in such a way, that the surface of the terrain is approximated optimally when the vertices are displaced by the height value. This can drastically improve the vertex count, compared to the uniform distribution, while keeping the visual fidelity nearly the same. The major drawback of TINs is the computationally expensive preprocessing required to generate them.

2.1.3 Right-Triangulated Irregular Networks

Other hybrid triangulation schemes mixing those two methods exist as well. These often reduce the preprocessing cost, while giving up only a marginal amount of accuracy.

One such triangulation scheme, often used for terrain rendering, is the **right-triangulated irregular network (RTIN)** [3] also known as **triangle bintree** [4]. The basic idea of this triangulation is the recursive subdivision of triangles alongside their longest edge, provided that they do not meet the lower bound of an error metric. A comparison between a uniform and an adaptive subdivision can be seen in Figure 2.2. This subdivision is referred to as **longest edge bisection (LEB)** and can be considered a constraint on the TIN triangulation. Due to their regular nature, the LEB subdivision can be computed in parallel, as will be discussed in Section 3.8.

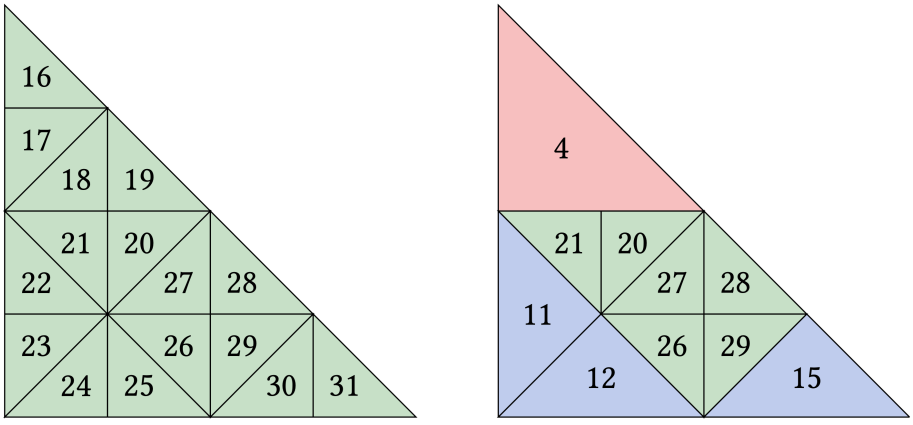


Figure 2.2: A comparison between a uniform and an adaptive recursive subdivision of a triangle alongside its longest edge. (taken from [3])

2.2 Level of Detail

Level of detail (LOD) algorithms [5] are performance optimizations. They select the complexity of an object depending on its contribution to the final image. The applications of these algorithms are vast and they can be utilized for textures and meshes alike.

To decide which LOD to use for rendering, multiple different error metrics can be chosen [5]. The simplest way is to select the level based on the distance to the camera on a logarithmic scale. Another more involved method is to measure the perceived error of the simplification in screen space, called the **screen-space error**.

2.2.1 Discrete Level of Detail

The traditional approach to LOD is called **discrete level of detail (DLOD)** [5]. As the name implies, the DLOD method renders the objects as one of multiple separate simplification steps. This is most commonly used for meshes, that have their simplifications precomputed, which are then dynamically chosen at runtime, as seen in Figure 2.3.

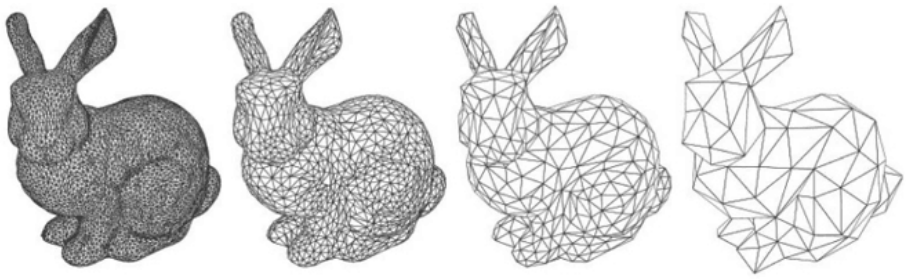


Figure 2.3: Four discrete levels of detail of the bunny mesh.
(taken from `http://robbinmarcus.blogspot.com/p/thesis.html`)

If the screen-space error between two adjacent LODs is not small enough to be imperceptible, the substitution of one object for the other will result in a visible **popping** artifact [5]. These undesirable discontinuities in video quality may be mitigated by **alpha blending** between the two levels, as depicted in Figure 2.4. Therefore the object is rendered at both LODs for a couple of frames, while the alpha value of the previous level fades to zero and that of the other fades to one. This smoothens the LOD switch but results in a higher render cost because the object has to be rendered twice during the duration of the transition.

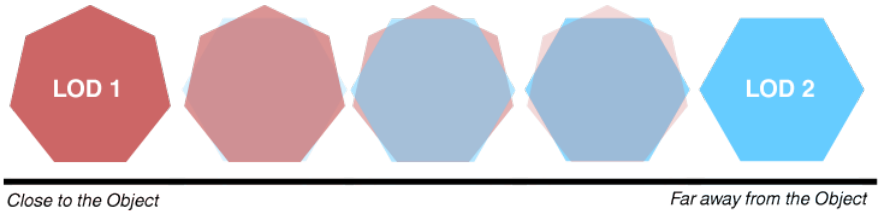


Figure 2.4: Two discrete levels of detail are interpolated using alpha blending.
(taken from `https://en.wikipedia.org/wiki/Popping_(computer_graphics)`)

2.2.2 Continuous Level of Detail

Terrains are commonly the largest object in a scene. They can span thousands of square kilometers but are covered with centimeter-sized details. Fortunately, the perspective projection of the camera compensates for the size differences, between different features, by making objects appear smaller, the further they are away from the viewer. The required screen-space detail of an object is inversely proportional to the distance between itself and the camera. Utilizing this characteristic for terrain rendering LOD solutions is crucial for meeting performance and memory constraints [6]. This implies that the terrain should not be represented by a single static mesh. Rather it should be updated continuously or stitched together from multiple smaller tiles of various resolutions and sizes. This thesis will refer to the former as **continuous level of detail (CLOD)** and to the latter as **multi-resolutional level of detail (MLOD)** from now on.

The notion of CLOD has multiple meanings in computer graphics literature. For one it simply describes level of detail (LOD) algorithms, that do not create individual discrete versions in a preprocess, but instead derive their representation from an algorithm-specific data structure at runtime [7]. Other literature [5] defines CLOD as an algorithm, which achieves a smoother transition between different detail levels than DLOD, by progressively simplifying a base mesh using vertex removals and half-edge collapses. This LOD scheme allows for fine-grained mesh simplification which can even be applied depending on the current view. In the following, this thesis will only refer to the former when mentioning CLOD. An illustration of such an LOD method can be seen in Figure 2.5.

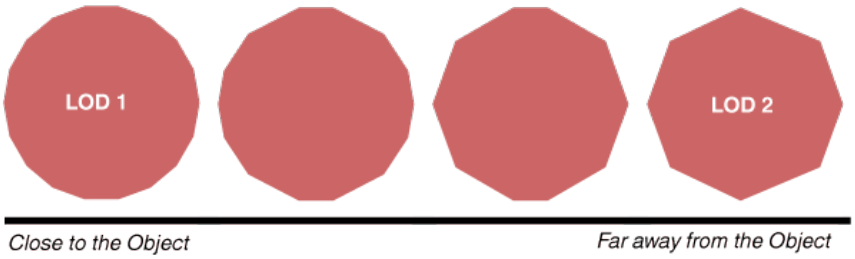


Figure 2.5: Continuous LODs filled in between their discrete counterparts (LOD 1, LOD 2). (taken from `https://en.wikipedia.org/wiki/Popping_(computer_graphics)`)

The most common CLOD method for terrain rendering is based on the LEB triangulation scheme mentioned previously [3, 4]. These algorithms recursively subdivide triangles covering the entire terrain alongside their longest edge, based on the screen-space error they would cause.

2.2.3 Multi-resolutional Level of Detail

As stated previously, MLOD refers to terrain surfaces, that are stitched together from multiple smaller tiles of various resolutions and sizes. This offers many advantages, especially with regards to **out-of-core (OOC) rendering** [8]. Some terrain data sets can not fit into memory all at once. They are required to be streamed in and out of VRAM. Therefore, they need to utilize OOC algorithms, which are specifically designed to operate with only a view-dependent subset of all terrain data.

The simplest MLOD approach is to split the terrain into fixed-size tiles, for each of which an individual LOD is selected [1]. This allows for a better view-dependent adaptation of the terrain than the DLOD equivalent. Compared to CLOD algorithms, the LODs of fixed-size tiles can not be determined based on the per-vertex screen-space error. Instead, they have to be computed based

on a less precise per-tile error metric. This decreases the accuracy, but in return requires far less LOD computations speeding up the algorithm. To improve the precision, one can precompute a local roughness map of the terrain [9]. With this, the available vertices can be distributed across the terrain, where they are needed the most. Using this method increases the terrain's size limit, but due to the fixed-size nature of the tiles, they all have to be loaded at once if the entire terrain should be visible at the same time [1]. It is also impossible to render triangles spanning multiple tiles at once, thus the algorithm may draw too many triangles, especially in the distance, where they may not be required.

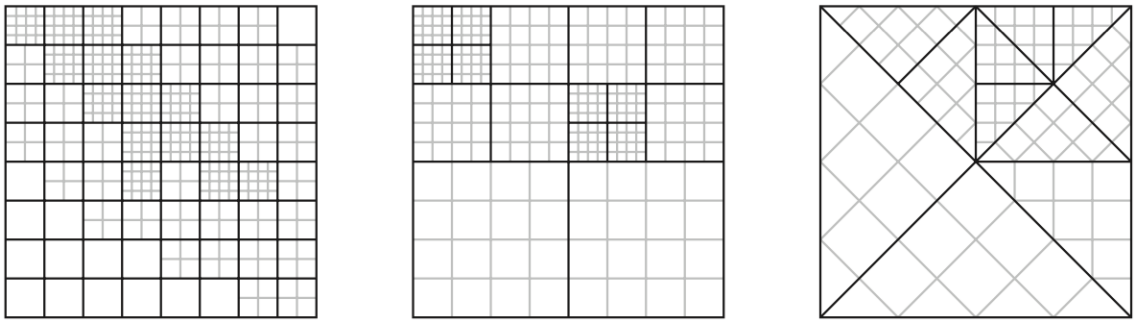


Figure 2.6: A comparison between three MLOD methods: fixed-size blocks on the left, quadtree subdivision in the middle, and binary tree subdivision on the right. The gray lines illustrate the subdivisions of different LODs. (taken from [1])

Natural extensions of the tile-based idea are methods that rely on hierarchical nesting, using a view-dependent **quadtree** or a **binary tree**, as can be seen in Figure 2.6. They are the most prominent algorithms in terrain rendering literature [6, 10, 11, 12]. The tiles of the former are divided into one or four rectangular tiles and that of the latter into one or two triangular tiles. Thus they decrease the size of their children by four and two respectively, while quadrupling/doubling the resolution. By utilizing this hierarchical structure, they address the major shortcoming of the fixed-size solution: the linear correspondence between the terrain size and the total amount of tiles.

Additionally, they are also inherently well suited for OOC rendering, due to their hierarchical structure and the independence of their tiles. But MLOD techniques also come with one inherent challenge - the meshes of the tiles, constituting the terrain, have to be seamlessly stitched together, to form one continuous surface [8]. Due to varying resolution, size, or triangulation of adjacent meshes, **gaps** in the geometry may arise. They are particularly distracting because they break the surface of the terrain and instead show the background (skybox, etc.) of the scene.

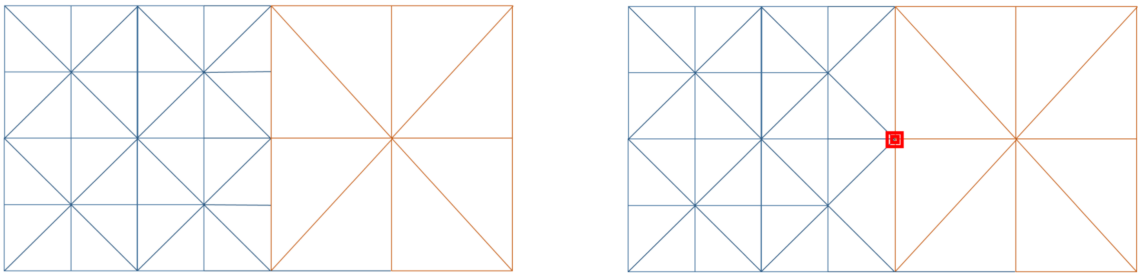


Figure 2.7: Showcases the t-junction problem between two adjacent LODs on the left and a possible stitching solution on the right. (taken from [12])

These gaps may be of two origins:

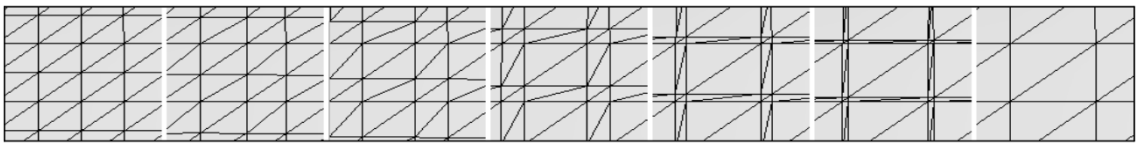
Cracks do occur when the vertices, at the edge between two tiles, do not fully align [6, 8]. This may happen, due to differences in the resolution of adjacent regular grids or differences in the triangulation of TINs and RTINs. For the latter, one solution constitutes the generation of the tiles in such a way, that the edge vertices share the same positions, with their potential neighbors. Alternatively, cracks can be masked using additional geometry, such as ribbons or skirts.

T-Junctions are a special form of gaps, commonly found in MLOD algorithms relying on regular grids [12]. As illustrated in Figure 2.7, if two adjacent tiles do not share the same LOD, some vertices of the one with greater vertex frequency will not align with that of its neighbor. T-junctions provide easier fixes than the general case, due to their regular nature. For example, the stitching used in the "Far Cry 5" terrain renderer simply shifts the position, of those vacant vertices, to their closest neighbors.

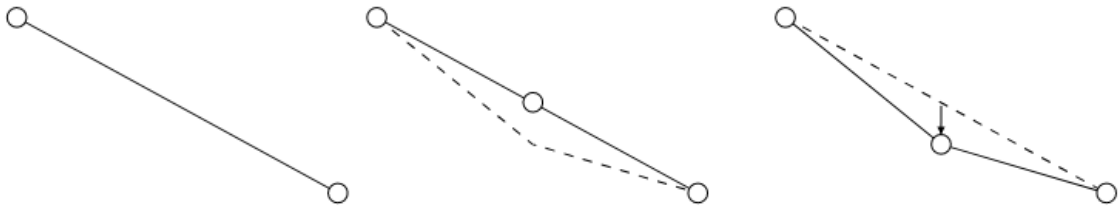
2.2.4 Level of Detail Morphing

No matter which LOD algorithm is used, for managing the terrain geometry, at some point vertices will have to be added or removed [6]. Since it is most often impossible to render a mesh detailed enough, such that these changes are not noticeable, the previously described undesirable pop-in artifacts will appear. To hide these usually small discrepancies, between the different versions of the mesh, there exist techniques that can animate changed vertices to move smoothly into their new positions. This process is called **vertex morphing** and can be applied to many CLOD and MLOD algorithms.

For regular grids, this technique is called **geomorphing** [6, 13] and works by interpolating between two regular grids, one of which possesses four times the resolution of the other. Every vertex of the higher resolution grid that does not align with a vertex of the lower resolution one, has to be interpolated between them, either vertically or horizontally, as seen in Figure 2.8.



(a) A grid as seen from above interpolated between two LODs. The odd vertices are morphed horizontally. (taken from [11])



(b) An edge as seen from the side interpolated between two LODs. Every other vertex is morphed vertically. (taken from [13])

Figure 2.8: A comparison of horizontal and vertical geomorphing.

2.3 Visibility Culling

Additional to LOD algorithms there exist another group of performance optimizations commonly used in computer graphics. Visibility culling [14, 15] is the process of removing objects or aggregates of primitives, which are not visible in the final image, before rendering them, as can be seen in Figure 2.9. Culling is traditionally performed on entire meshes by the CPU and on individual triangles after passing the vertex shader. But in recent years, with the advent of compute shaders and indirect draw calls, the culling in major graphics engines is more and more shifted to the GPU, due to its immense parallel processing power. In the following, the four most common visibility culling methods will be presented.

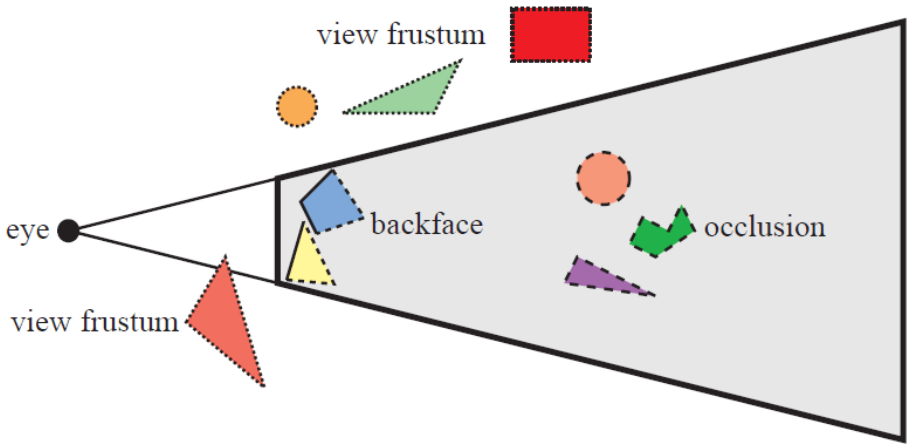


Figure 2.9: Showcases different forms of culling. All dotted surfaces do not contribute to the rendered image and can be culled using either view frustum, backface, or occlusion culling. (taken from [14])

2.3.1 View Frustum Culling

View frustum culling [14] describes the process of skipping the rendering of objects or primitives, which lie outside the view frustum of the camera and thus are not visible in the final image. For complex geometry, a conservative approximate bounding volume is typically used to speed up the culling. Additionally, a hierarchal structure organizing the bounding volumes called the bounding volume hierarchy can be utilized to cull multiple objects at once. Furthermore, the graphics pipeline incorporates clipping, a method that automatically discards triangles that lie entirely outside the view frustum, before the rasterization stage.

2.3.2 Backface Culling

Backface culling [14] denotes methods, that omit the rendering of surfaces or triangles, which face entirely away from the camera. This process is also automatically performed by the graphics pipeline on a per-triangle basis after they have passed the vertex shader. Additionally, there exist clustered backface culling algorithms, used to cull entire aggregates of triangles efficiently, by collecting a minimal cone from their vertex normals and testing whether it faces away from the viewer.

2.3.3 Z-Buffering

In a broader sense, the visibility technique called **Z-buffering** [14], used to ensure that fragments closer to the viewer properly occlude those further away, can be considered a culling method as well. It is performed on a per-fragment basis after rasterization and compares the distance to the camera with the current value of the **z-buffer** texture, which stores the depth information of previously rendered fragments. If the fragment is further away than the previous one, it is invisible and thus it will be discarded.

2.3.4 Occlusion Culling

Occlusion culling [14] algorithms utilize the fact that objects, entirely concealed behind other opaque geometry, do not contribute to the rendered image as well and may be culled. Commonly, occlusion culling algorithms are based on the following three steps. At first, a hierarchical **z-pyramid**, an extension of the classic z-buffer, which is mipmapped using a maximum filter, is generated based on some of the best occluders of the scene. For each object, which is tested for occlusion, the bounding volume is projected into screen space and the appropriate mip level of the pyramid is determined. Finally, the depth values of said object and the mip level are compared and thus occlusion can be detected.

2.4 Hardware Tessellation

Since the early 2010s, a new extension to the traditional graphics rendering pipeline called hardware tessellation exists [14]. As can be seen in Figure 2.10, it is a technique that can introduce additional vertices into triangles or quads of a base mesh, depending on a customizable tessellation factor, which is calculated based on supplementary surface information such as heightmaps or curvature data.

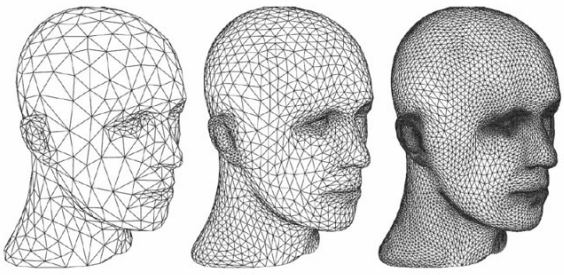


Figure 2.10: The base mesh on the left is tessellated to different degrees on the right. (taken from https://rastergrid.com/blog/2010/09/history-of-hardware-tessellation)

Hardware tessellation extends the pipeline with two new shader stages [14], the **control shader**, and the **evaluation shader** as well as the fixed function **tessellator** sitting in between. The entire traditional graphics rendering pipeline can be seen in Figure 2.11. The control shader operates on so-called patches (quads, triangles, lines) and is executed after the vertex shader. Its role is to determine the tessellation factor, which has to be calculated based on an LOD criteria, as well as additional attributes required by the tessellator. After that, the tessellator subdivides the patches according to the information supplied by the control shader. Finally, the programmable evaluation shader is responsible for transforming the newly obtained points into vertices by applying displacement, which is based on the surface information of the object.

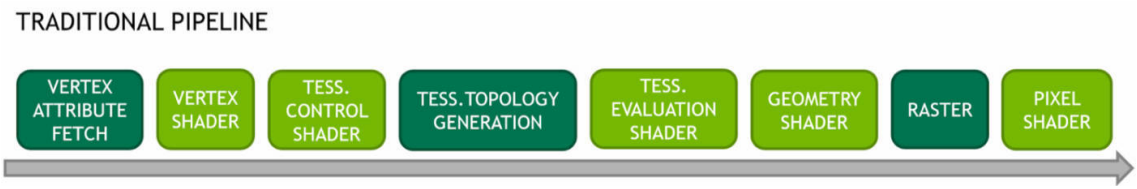


Figure 2.11: An overview of the traditional graphics pipeline. It is executed per draw call from left to right. (taken from `https://khronos.org/blog/mesh-shading-for-vulkan`)

Hardware tessellation combines triangulation and LOD into one parallel hardware accelerated approach [9]. Due to its supplementary nature and reliance on a base mesh, it is commonly used as an optimization on an existing terrain rendering algorithm and works especially great in combination with MLOD algorithms.

Although most modern desktop devices do support hardware tessellation, this technique is still not available on most mobile devices and older machines. Currently, tessellation is also not part of the specification of the WebGpu API [16], which will be used to implement the rendering techniques of this thesis.

2.5 Terrain Data

Many terrain rendering papers only cover the triangulation as well as the LOD algorithm of the geometry, but most real-world implementations must solve an additional problem: How should the terrain data be represented? As can be seen in Figure 2.12, it is common to store additional data [12] covering the terrain in two dimensions, such as albedo, normal, shading, and use-case-specific information. Because this information is often not only required for rendering the terrain itself but also in other shaders or systems, it is important to store this information in a data structure that can be easily looked up at any location. With these requirements in mind, it is not feasible to store the entire terrain data as vertex attributes on the terrain mesh/meshes itself, but it rather requires separate consideration.

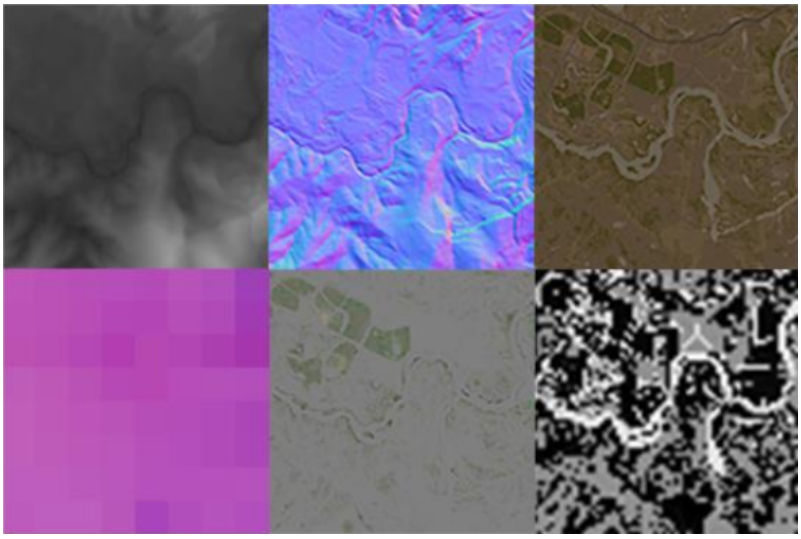


Figure 2.12: An example of six terrain textures used in the game "Far Cry 5". (taken from [12])

The GPU native structure for two-dimensional random access is called a **texture**. Utilizing one texture per different terrain data type is a valid and commonly practiced approach for small or low-detail terrains. This technique, however, quickly runs into scalability problems for very large or detailed landscapes. Similar to the geometry, an LOD strategy has to be utilized to achieve terrain data coverage across multiple orders of magnitude of difference in scale. Additionally, this strategy should support OOC rendering not to be limited by the amount of available system memory.

Because textures can not vary their accuracy the same way meshes can, CLOD techniques can not be applied, but the textures have to be represented at multiple resolutions instead. This is most commonly done by storing the data as tiles of the same texture size, which cover the terrain at different detail scales.

2.5.1 Quadtree

One suitable hierarchical structure for containing these tiles is the quadtree, shown in Figure 2.13. As seen in Section 2.2.3, they are great at partitioning mesh data as well. Some implementations [11, 12] even use the same quadtree for handling the LOD of both the meshes and the terrain textures simultaneously. This is why the quadtree is a favored data structure used to represent terrains.

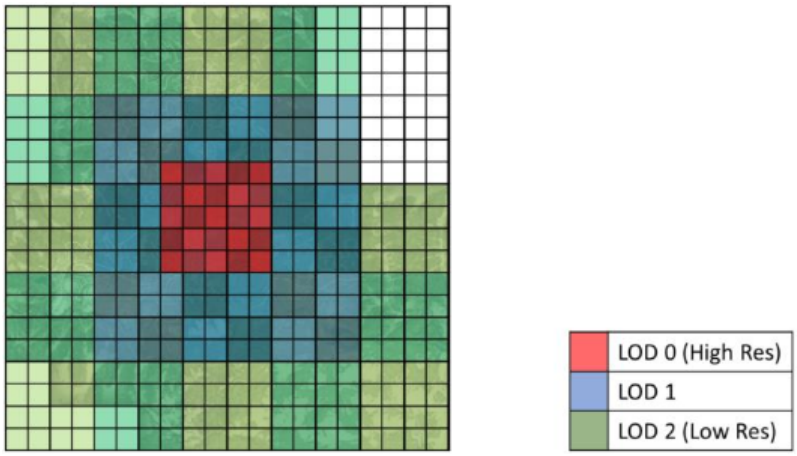
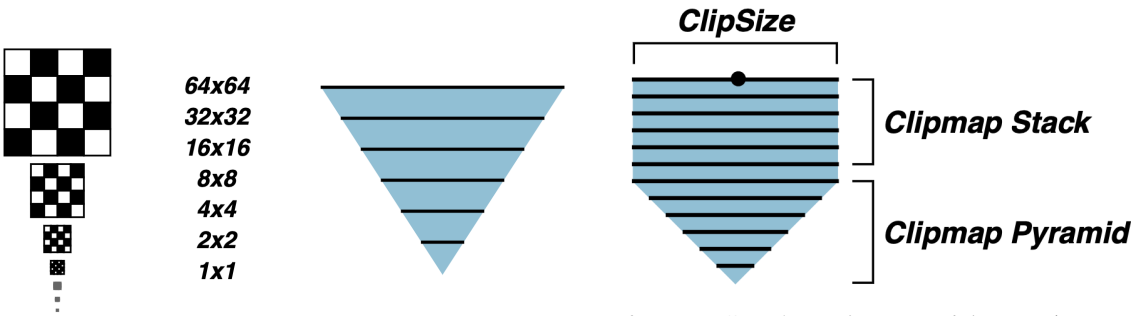


Figure 2.13: A quadtree with a depth of three. The red tiles are the tiles with the highest and the green ones with the lowest resolution. (taken from [12])

2.5.2 Clipmap

The other major candidate for efficient large-scale terrain data storage is the **clipmap** [17]. It is a data structure that can efficiently store a view-dependent subset of an arbitrarily sized texture in memory. The clipmap is a natural extension of the mipmap in which not all data resides in VRAM at once. This is achieved by clipping each mip layer to a maximum size called the *clipsize*. Consequently, the texture pyramid, compared to a conventional mipmap, is divided into a clipmap stack, consisting of layers with $clipsize^2$ texels and a clipmap pyramid storing the portion of the mipmap smaller than the *clipsize*, as seen in Figure 2.14. Clipmaps limit the exponential increase in texture size of conventional mipmaps per mip layer to a merely linear one. This allows for caching the currently relevant portion, of petabytes of source data, using only a few megabytes of memory.



(a) A mipmap pyramid on the left and its schematic side-view diagram.

(b) A side-view diagram of a clipmap consisting of a stack and a pyramid.

Figure 2.14: A comparison between a conventional mipmap on the left and a clipmap on the right. (taken from [17])

Because of their view-dependent nature, clipmaps must always be up-to-date, according to the current viewer position, during rendering [17]. This is accomplished by updating the clipmap stack from frame to frame. Due to the temporal consistency of the view position, also called the clipmap center, most of the data cached at each layer of the clipmap will stay the same. Utilizing this fact, only a small border has to be updated, which is achieved by addressing the texture layer toroidally (e.g. wrapping the texture coordinates modulo the *clipsize*). Figure 2.15 shows this process: as the clip center starts roaming across the source image, the clipmap stack is updated in place.

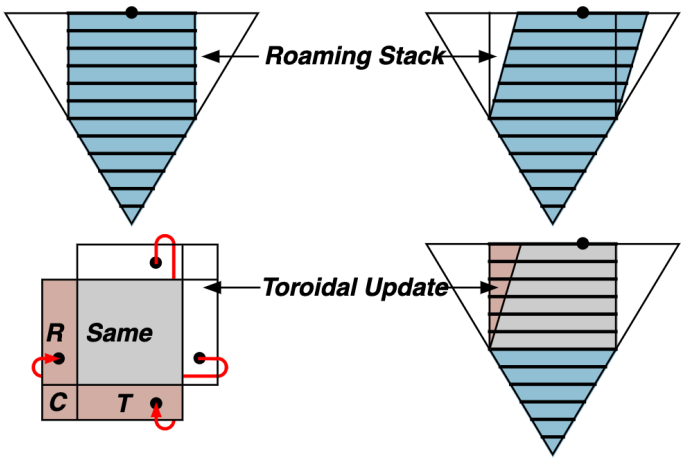


Figure 2.15: Visualizes the process of toroidal updating the clipmap according to the clipcenter. (taken from [17])

Additionally, C. Tanner et al. [17] the authors of the original clipmap paper describe the virtualization of clipmaps for numerical ranges exceeding floating-point address ranges and dynamic clipmap management with the main memory as a second-level cache.

The authors [17] presented the clipmap as a data structure that is implemented in the graphics API and built into the low-level hardware. Unfortunately, that has not happened, which is why not all details of their technique apply to modern terrain rendering. Especially their texture data addressing scheme can only be emulated in software in the fragment shader and is not hardware accelerated by a clipmap-specific texture sampler.

2.6 Real-World Data

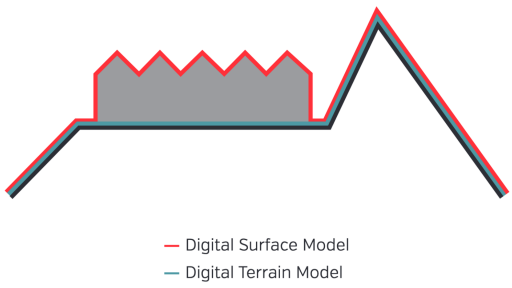
The terrain data can have one of two origins, either it is synthesized using specialized software and sculpted by an artist, or it is measured from the real world, using specialized hardware like lidar sensors, or high-resolution cameras.

2.6.1 Digital Elevation Model (DEM)

The heightmap is the fundamental information required by every terrain. When measuring real-world landscapes, either the **digital terrain model (DTM)** [18] or the **digital surface model (DSM)** data can be obtained. The former, also known as **digital elevation model (DEM)**, represents the bare-earth surface of the terrain, excluding any objects, buildings, or vegetation. While the latter refers to the surface including all of the said objects. The difference is illustrated in Figure 2.16.

2.6.2 Digital Orthophoto (DOP)

Another important data type used to visualize terrains is the albedo information, which determines the surface color of the terrain at any position. **Digital orthophotos (DOPs)** [18] are satellite or aerial images, which have been corrected for the distortion introduced by the surface of the terrain, such that the rooftop and the base of every building align vertically and no sidewalls are visible. An example of such an DOP can be seen in Figure 2.16.



(a) Shows the difference between DTM (surface without objects) and DSM (surface including objects above). (taken from [18])

(b) An orthophoto of a portion of the town of Hartenstein. (taken [19])

Figure 2.16: Showcases a digital elevation model (DEM) schematic on the left, and a digital orthophoto (DOP) on the right.

Chapter 3

Previous Work

Over the last few decades, a multitude of different terrain rendering algorithms, that adapted to the hardware features and capabilities of their time, have been developed. In the following, the author will survey a prominent subset of them in chronological order. Therefore the author will heavily rely on the previously introduced terminology of Chapter 2. Figure 3.1 shows a comparison between the covered techniques, regarding a few central characteristics, which should serve as a quick overview and orientation.

Algorithm	Displacement	Triangulation	LOD	Culling	Terrain Data	Year
ROAM	CPU	RTIN CPU Run-Time	CLOD CPU	Frustum CPU	Not specified	1997
GeoMipMap	CPU	Grid CPU Preprocess	MLOD CPU Quadtree	Frustum CPU	Not specified	2000
Chunked LOD	CPU	RTIN CPU Preprocess	MLOD CPU Quadtree	Frustum CPU	Quadtree	2002
GeoClipmaps CPU	CPU	Grid CPU Run-Time	MLOD Clipmap	Frustum CPU	Clipmap	2004
GeoClipmaps GPU	GPU	Grid CPU One-Time	MLOD Clipmap	Frustum CPU	Clipmap	2005
PGM	GPU	Grid CPU One-Time	DLOD	None	Clipmap	2007
CDLOD	GPU	Grid CPU One-Time	MLOD Quadtree	Frustum CPU	Not specified	2010
Far Cry 5	GPU	Grid CPU One-Time	MLOD GPU Quadtree	Frustum Back-face Occlusion GPU	Quadtree	2018
CBT	GPU	RTIN GPU Run-Time	CLOD GPU	Frustum GPU	Not specified	2020
UDLOD	GPU	Grid GPU Run-Time	MLOD GPU Quadtree	Frustum GPU	Chunked Clipmap	2022

Figure 3.1: A central comparison between a selection of influential terrain rendering algorithms. The UDLOD method presented in this thesis is located at the bottom of the table.

3.1 Real-time Optimally Adapting Meshes (ROAM)

In 1997 M. Duchaineau et al. [4] introduced the **Real-time Optimally Adapting Mesh (ROAM)** terrain rendering method [4], which generates a mesh triangulation by splitting and merging triangles according to a screen-space error metric. The algorithm produces optimal triangle bintrees (RTIN), which are updated frame to frame to maintain a continuous surface, as can be seen in Figure 3.2. Additionally, vertex splits and merges are animated (see Section 2.2.4) to ensure temporal consistency. Furthermore, multiple performance enhancements like triangle-based view frustum culling and progressive mesh optimizations are described.

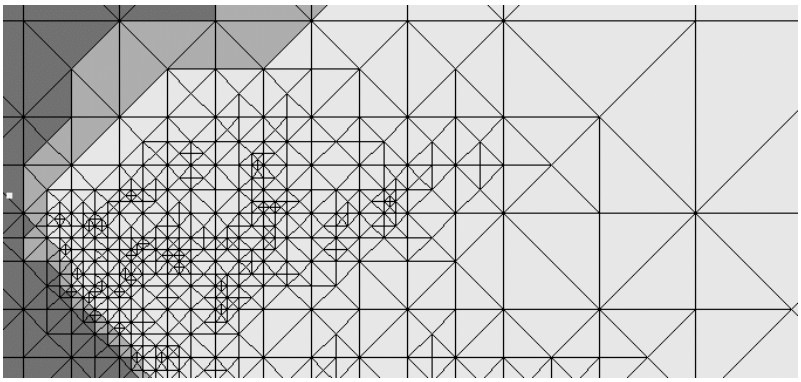


Figure 3.2: Depicts the triangulation of the ROAM algorithm without height displacement from a top-down view. (taken from [4])

Although continuous level of detail (CLOD) methods [4, 7] similar to ROAM deliver an optimal view-dependent triangulation of the terrain they have fallen out of favor, since the advent of powerful consumer GPUs in the early 2000s, because of their sequential nature and their high CPU overhead. These methods occupy too much of the processing power of the CPU, especially for large frame resolutions while maintaining a low screen-space error threshold. Additionally, frequent updates to the terrain mesh while the viewer is moving, cause a lot of vertex data uploads to the GPU [10, 6]. These frequent uploads are often more expensive than rendering more triangles with a less optimal distribution that already reside in VRAM. Regardless, there have been attempts in recent years to parallelize the LEB triangulation on the GPU [3] (see Concurrent Binary Trees in Section 3.8).

3.2 Geometrical MipMapping (GeoMipMap)

In comparison to ROAM the **Geometrical MipMapping** approach outlined by Willem H. de Boer [10] better utilizes the graphics hardware by "pushing as much triangles through the pipeline as [it] can handle, with the least amount of CPU overhead" [10]. Therefore Boer proposes an MLOD method, which divides the terrain into a quadtree of small regular grid tiles, which can be rendered at different resolutions similar to texture mipmaps, as seen in Figure 3.3. The appropriate GeoMipMap level for each tile is chosen based on the screen-space error the level introduces compared to its original (highest resolution).

Additionally, Boer [10] introduces a method for loading only the currently required GeoMipMap levels of the tiles into memory, thus enabling OOC rendering of large datasets. This algorithm is executed on the CPU and can be sped up by using look-up tables for the errors.

Figure 3.3: Shows the different levels of mesh simplification of the GeoMipMap algorithm. (taken from [10])

The geometry gap problem caused by the MLOD is solved by using specific index lists for the edges of the grids, which omit every other vertex to match with the lower LOD neighbors [10]. Furthermore, it is possible to decrease the visual discrepancy between adjacent grids of differing resolutions by interpolating between two LODs, similar to trilinear texture filtering.

3.3 Chunked Level of Detail

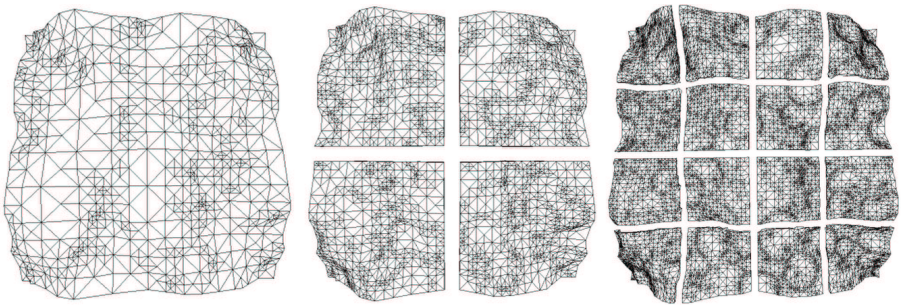


Figure 3.4: Depicts the first three tree levels of a chunked LOD terrain. (taken from [6])

Thatcher Ulrich [6] introduced **Chunked Level of Detail**, another similar MLOD algorithm based on a quadtree. This algorithm relies on a tree of independent preprocessed meshes (RTIN) called chunks which are assigned a maximum object-space geometric error, from the original base mesh, per tree level. The correct LOD of the chunks is chosen at run-time purely based on the geometric error and the distance to the point of the bounding volume closest to the viewer, by traversing the quadtree from the root. An example of such a tree can be observed in Figure 3.4. To avoid the popping issue introduced by MLOD algorithms he proposes a vertical morphing of the chunks with their parents. Ulrich solves the crack problem by extending each edge with a vertical skirt pointing downwards, as shown in Figure 3.5. This hides the holes between neighboring chunks and is unnoticeable for low screen-space errors.

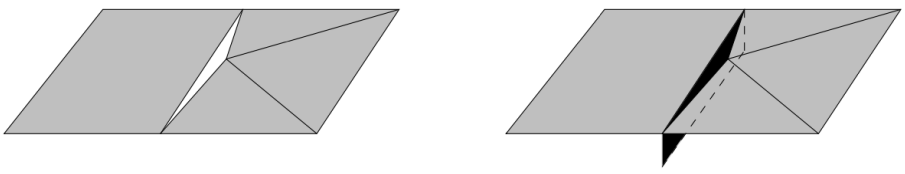
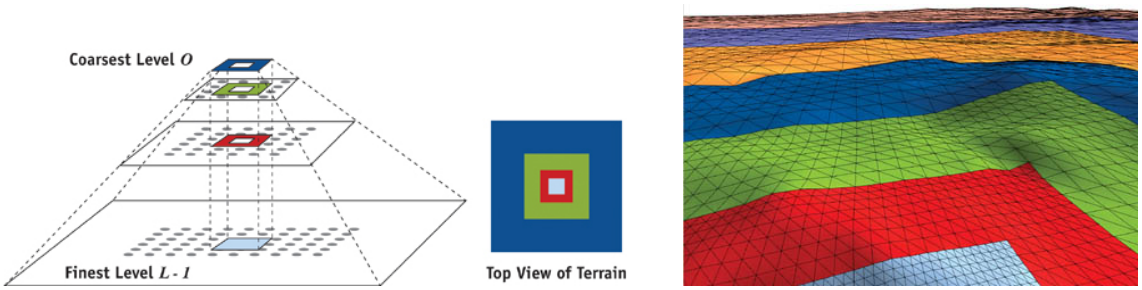


Figure 3.5: The vertical skirt (black rectangle) is used to fill the cracks between two neighboring chunks. (taken from [6])

3.4 Geometry Clipmaps

One of the most popular terrain rendering methods is called **Geometry Clipmaps** and was introduced by Hugues Hoppe and Frank Losasso [20]. They represent the terrain mesh as a stack of nested regular grids centered under the viewer, which increase in size as they span outward. Unlike the previous algorithms, geometry clipmaps do not adapt to the screen-space error, but instead, assume the worst-case terrain with uniform detail and thus only consider the distance to the viewer for the LOD calculation. This results in a nearly uniform and pixel-sized triangulation in screen space, which they claim can be rendered efficiently at 60 frames per second (FPS).

The vertex data is stored in vertex buffers that are accessed toroidally, which means that reads and writes are carried out modulo the size of the grid [20]. This enables efficient incremental updates to the mesh as the viewer moves about the terrain. Additional terrain data is cached in texture clipmaps, a pyramid of equally sized textures, which cover the data at multiple scales similar to mipmaps, as can be seen in Figure 3.6. By that, their approach unifies the LOD of the terrain's geometry with that of its additional terrain data.



(a) Shows how the different clipmap level project into world space.

(b) A terrain rendered using Geometry Clipmaps.

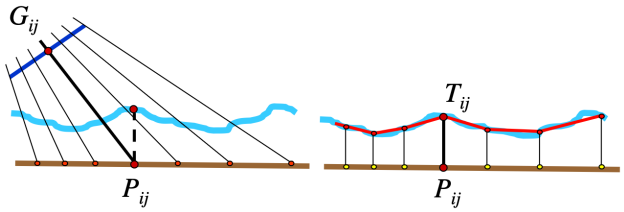
Figure 3.6: Showcases the Geometry Clipmaps algorithm, in which the mipmap pyramid on the left corresponds to the rendered terrain on the right. (taken from [21])

To mitigate cracks at the boundaries of the nested grids, they insert vertical zero-area triangles to hide the existing T-junctions [20]. Also, mesh morphing and custom texture blending can be implemented, for a smooth spatial and temporal continuous LOD transition. Furthermore, geometry clipmaps allow for incremental terrain data decompression as well as detail synthesis from fractal noise.

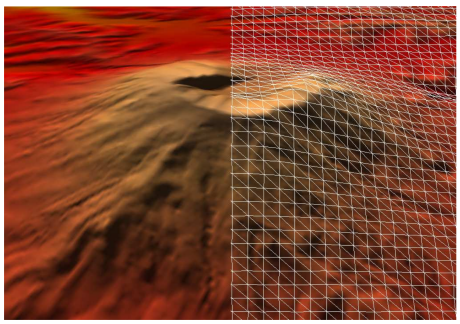
In a later iteration of their algorithm called **GPU-Based Geometry Clipmaps**, Hoppe [21] describes an implementation, which utilizes the, at that time recently introduced, unified shader cores of modern GPUs, which provide the ability to sample textures in the vertex shader. This allows for applying the height displacement in the vertex shader instead of having to update the grid meshes on the CPU. Therefore the height information is cached as a clipmap as well, merging the geometry and texture data representation. Consequentially the same meshes can be used to render each layer of the geometry clipmap, resulting in index and vertex buffers that do not need to be updated after initialization.

3.5 Persistent Grid Mapping (PGM)

In 2007 Yotam Livny et al. [22] presented the **Persistent Grid Mapping (PGM)** framework for rendering large-scale terrains. There they tessellate the terrain using a uniform screen-space persistent regular grid, which resides in GPU memory. This grid is then projected onto the terrain base plane and displaced by the height value in the vertex shader, resulting in a view-dependent approximation of the terrain's surface, while ensuring the absence of cracks or other unwanted artifacts. The projection and the resulting tessellation can be seen in Figure 3.7.



(a) Shows the mapping of the persistent grid onto the terrain base plane on the left and the subsequent height displacement on the right.



(b) A terrain rendered using PGM, with a low-resolution wireframe view layered on top.

Figure 3.7: Showcases the Persistent Grid Mapping algorithm, as a schematic on the left and as a rendered image on the right. (taken from [22])

As can be seen on the left of Figure 3.8, for flat camera angles mountain peaks close to the camera may be missed by the projection, due to the offset of the sampled region. To resolve this problem an auxiliary camera is employed that is only used for sampling the terrain height and generating the surface geometry. This camera utilizes a similar view frustum as the primary camera with the deciding difference that the angle between its center and the horizontal plane is shifted downwards. This is done in such a way that the entire sampled region of the base plane, starting under the viewer and ending at the horizon, lies inside the frustum. This guarantees that the entire visible region is contained in the sampled one.

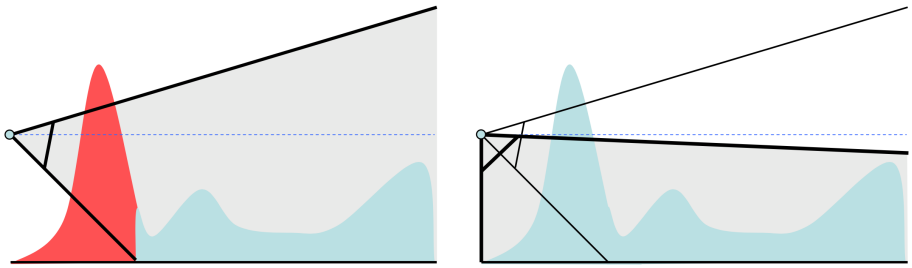
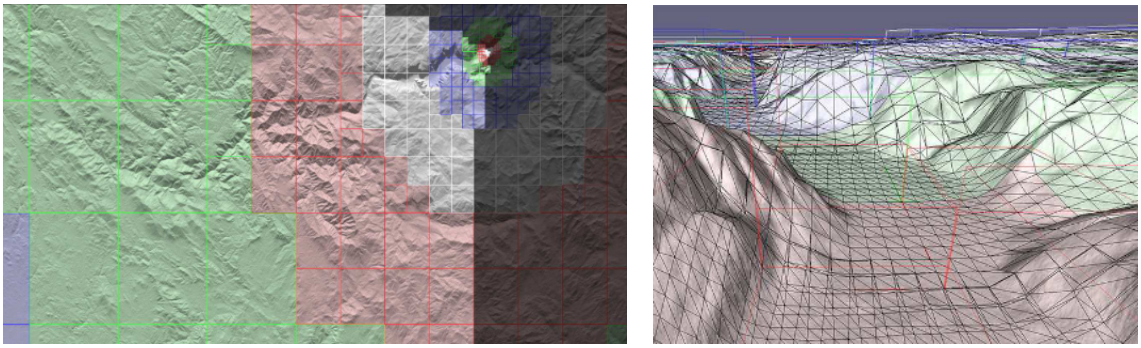


Figure 3.8: Visulaizes the camera restriction. On the left, a peak (in red) is missed near the camera due to the mismatch of visible and sampled regions. On the right, this problem can be rectified by utilizing an auxiliary camera.

This algorithm automatically guarantees a seamless LOD of the terrain geometry, that transitions smoothly, without requiring any preprocessing [22]. PGM does not rely on any explicit culling methods, because the grid already covers the entire view frustum. Although this means that vertices that are occluded still have to be processed. Additionally, they present an OOC terrain data streaming solution, which is based upon clipmaps, as introduced in Section 2.5.2.

3.6 Continuous Distance-Dependent Level of Detail (CDLOD)

The paper "**Continuous Distance-Dependent Level of Detail (CDLOD)** for Rendering Heightmaps" by Filip Strugar [11] claims to be a refinement of both the Chunked LOD (Section 3.3) and the Geometry Clipmaps (Section 3.4) algorithm. It combines the efficient and scalable quadtree data structure of the former with the vertex-shader height map displacement of the latter. The goal of CDLOD is to distribute as many triangles as possible evenly across the screen, without relying on screen-space error metrics. Therefore the mesh is organized as a quadtree of heightmap tiles, which are triangulated using regular grids. The LOD of these tiles is determined discretely and solely based on their logarithmic three-dimensional distance to the viewer. It is calculated for each frame during a top-down traversal of the tree, while frustum-culling them, which speeds up the rendering as well as the traversal itself. Tiles can only be partially selected at the border between adjacent LODs, which allows for earlier and more flexible transitions between them while increasing performance.



(a) Shows the LOD selection of quadtree nodes for rendering. The nodes in the dark area are frustum culled.

(b) Depicts the morphing of the meshes across the LODs.

Figure 3.9: Showcases the improvements of the CDLOD algorithm over previous MLOD techniques. (taken from [11])

The popping and cracking issues caused by the multi-resolutional geometry are solved jointly by horizontally morphing tiles adjacent to the LOD border [11], as discussed in Section 2.2.4. This is achieved by pushing every other vertex toward its neighbor depending on a morph factor, which is calculated based on its distance to the viewer, resulting in a completely continuous mesh. One drawback of this algorithm is its inability to morph between adjacent tiles, with LODs varying by more than one. This limits the minimum possible quadtree depth or the viewing distance.

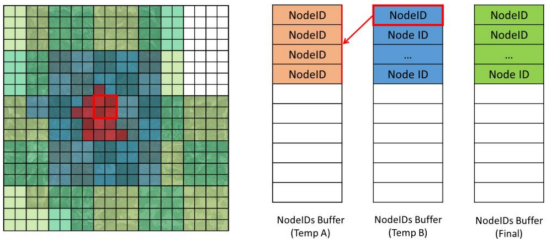
3.7 Far Cry 5 Terrain Renderer

Another more recent entry in the elaborate list of terrain rendering algorithms is the system used in the game called "**Far Cry 5**" developed by Ubisoft Montréal [12]. It is an interesting and important addition to this survey, because of its great interplay with other shaders and systems of the game. This is an important insight that many of the aforementioned methods do not consider. The game's terrain is based on a quadtree, that both manages terrain data and the LOD of its geometry, which consists of regular grid tiles. In their described implementation they focus on shifting as much work as possible away from the CPU, resulting in the quadtree traversal, culling, and rendering being entirely executed on the GPU. Because of that, the team could even reduce the GPU cost of the terrain renderer, by maximizing the vertex culling performed in a compute shader, while also making the terrain data available for other systems such as vegetation placement.

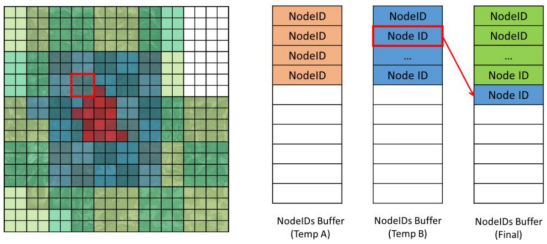
The described algorithm requires seven key GPU data structures:

- The **node atlas** storing the data of the currently loaded nodes of the quadtree in a texture atlas.
- The **node description buffer** storing information (min/max height, LOD bias, node atlas index) of the currently loaded nodes.
- The **quadtree texture** holding the index into the node description buffer for every node of the entire terrain.
- The **node list** buffer build during the quadtree traversal of each frame, storing the quadtree tiles that should be rendered.
- The **lod map** recording the LOD of each sector (smallest tile) of the terrain in an easy-to-lookup texture.
- The **sector map** recording the node atlas ID of each sector (smallest tile) of the terrain in an easy-to-lookup buffer.
- The **patch list** buffer, which is a partitioned, refined, and culled version of the node list.

The terrain data streaming is the only part of the renderer that is not implemented on the GPU [12]. It is responsible for updating the quadtree based on the viewer's position. Therefore the CPU uploads newly loaded node data into the node atlas and sends updates to the quadtree texture and node description buffer accordingly, notifying them of the current quadtree state.



(a) The four child nodes are loaded. Thus split the node into four children and process them again on the next layer.

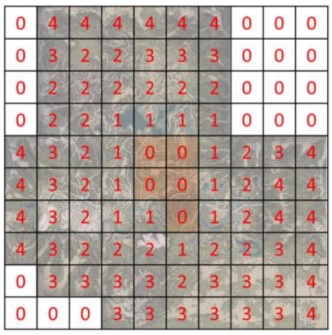


(b) Not all four children are loaded, thus this node currently is the best representation of the area and should be used for rendering.

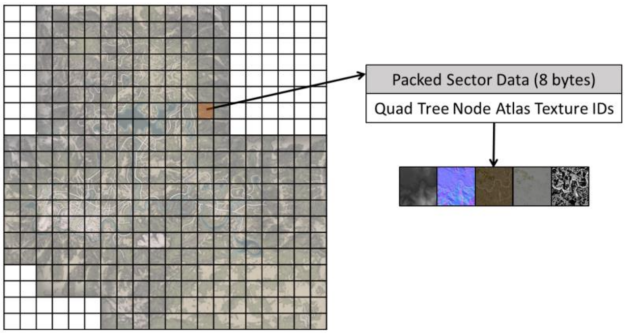
Figure 3.10: An explanation of the node list building algorithm. The left node can be split, while the one on the right is already the best currently loaded version.
(taken from [12])

Using that information, every frame, by traversing the quadtree layer by layer in a compute shader, the node list, storing the node IDs (indices into the node description buffer) selected for rendering is built [12]. This process starts at the root of the tree by filling one of two temporal node buffers with the root-level nodes. Then all of these nodes are either appended to the second temporal buffer if all four child nodes are loaded as well or to the final node list otherwise. This process can be seen in Figure 3.10. After all nodes have been processed, the two temporal buffers are swapped and the procedure is repeated for the next layer.

Subsequently, the lod and the atlas map, illustrated in Figure 3.11, are generated based on the final node list by means of another compute shader [12]. The atlas map is a convenient lookup data structure for accessing the terrain data of the node atlas, at any position inside the terrain.



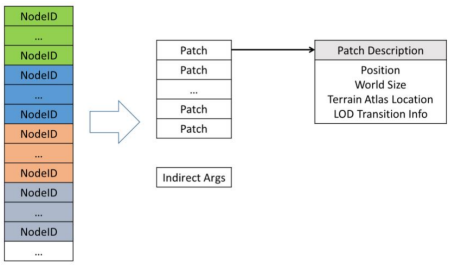
(a) The lod map, which stores LOD of each sector.



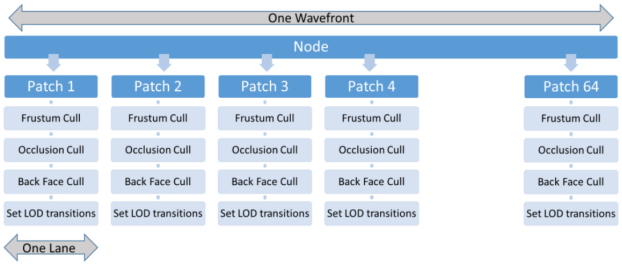
(b) The atlas map, which stores a description of all atlas IDs covering the sectors.

Figure 3.11: Illustrates the lod and atlas map generated each frame based on the node list.

Before finally rendering the terrain mesh a final data structure - the patch list - has to be populated for each view [12]. This buffer contains the information of all terrain patches that will be rendered using one instanced draw call and a single regular grid mesh. Therefore each node of the node list is split into 8 by 8 patches, which are frustum, occlusion, and backface culled in parallel as part of a single wavefront. This can be seen in Figure 3.12. Additionally, the LOD information of adjacent nodes is looked up in the lod map, during this compute shader execution, which will later be used for stitching. This highly parallel and efficient algorithm produces a view-optimized terrain mesh which can be rendered at low cost due to the aggressive culling.



(a) The nodes are split up into patches and rendered using an indirect argument buffer.

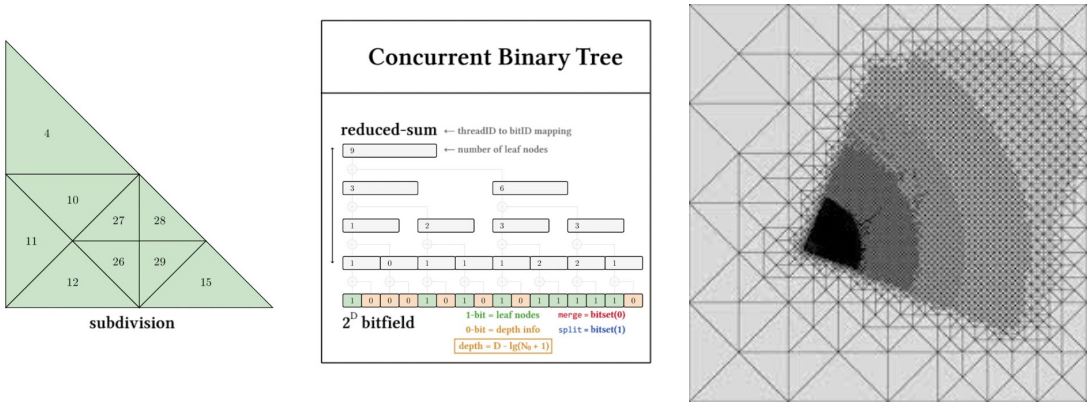


(b) Shows the node-to-patch refinement which is executed as part of a single wavefront per patch.

Figure 3.12: Shows the process of turning the node list into the patch list. (taken from [12])

3.8 Concurrent Binary Trees (CBT)

One of the most recently published terrain rendering techniques is called "**Concurrent Binary Trees (CBTs)** (with application to longest edge bisection)" [3]. The author Jonathan Dupuy introduces the CBT data structure, which as he claims can accelerate any tree-based subdivision algorithm such as LEB by computing it in parallel. The CBT is represented as a binary heap (a 1D array) with a size of 2^{D+1}, where D indicates the depth of the tree. The first 2^D elements of the heap store the sum of their two respective children, while the second half implicitly describes the configuration of the tree, where elements set to one denote the presence of a leaf node. An overview of the CBT algorithm is illustrated in Figure 3.13.



(a) An overview of the CBT algorithm. The second half of the 2^{D+1} bitfield can fully encode any subdivision of depth D on the left.

(b) Showcases the subdivision achievable by the CBT algorithm.

Figure 3.13: Shows the data structure on the left and the achievable subdivision on the right of the CBT algorithm. (taken from [3])

This data structure is then used to evaluate the longest edge bisection (LEB) subdivision scheme in parallel on the GPU, greatly increasing performance compared to the similar ROAM algorithm, described in Section 3.1, which performs the subdivision on the CPU [3]. This is also the main benefit of CBTs, namely the linearly increasing processing speeds with respect to the number of processors running in parallel.

The retained nature of the data structure implies that the entire tree has to be stored in memory, which can get exceedingly large for growing terrain sizes. Besides, the binary tree can only merge or split one adjacent triangle at a time, which limits its adaptability for fast-paced camera movement. On top of that, a major drawback lies within the binary nature of the edge information. Animating the vertex splits and removals is an unsolved problem for the CBT algorithm, which necessitates a screen-space error of less than one pixel for each of them, to mitigate the issue of vertex popping.

Chapter 4

A Novel Terrain Rendering Method

Having summarized and outlined the important innovations of previous terrain rendering algorithms, the author will now evaluate their successes and shortcomings. Therefore the author will at first present what he considers an "ideal terrain renderer" and then discuss the tradeoffs of the presented methods. Together with the knowledge of the previous two chapters as a foundation, finally, the author proposes a new terrain rendering method that attempts to satisfy all of the eight requirements of the "ideal terrain renderer" simultaneously.

4.1 The Ideal Terrain Renderer

The "ideal terrain renderer" described in this section features eight general requirements, which the author deems a general-purpose terrain renderer should satisfy. Of course, not all real-world applications incorporating terrain rendering do necessitate all of these properties.

1 Hardware Adaptive Quality First of all, it is important to state that any terrain renderer is limited by the processing power, memory, and capabilities of the executing hardware. While these can vary drastically from device to device it is crucial that the quality of the algorithms can be adjusted to achieve a consistent frame rate on most modern consumer hardware.

2 Unlimited Terrain Size The terrain size should not be constrained by the limitations of the data structures and algorithms used, but only by the disk space required to store the data. It should be possible for the algorithm to render landscapes as large as planet earth, with a resolution of up to one meter in real-time. This implies that the rendering method should efficiently scale across up to seven orders of magnitude (or 25 powers of two).

3 Seamless Viewing Distances The terrain should be viewable from any reasonable distance while maintaining its visual fidelity relative to it. The transition between viewing the landscape from afar and on the ground should be seamless regarding visual quality and frame rate.

4 Uniform Screen-Space Error Although most devices will not allow for approximating the terrain geometry without any screen space error, it is important to keep the error consistent across the final image and in between frames.

5 Smooth Vertex Morphing Accepting that an imperceptible screen space error is not feasible, the impact of such imperfections (popping) should be minimized by distributing the discrepancy over multiple frames. This can be achieved by morphing (see Section 2.2.4) between vertex positions.

6 Random-Access Terrain Data The terrain data should be accessible from other systems than the terrain rendering algorithm itself, while allowing for the retrieval of the information at any position of the terrain with view-dependent accuracy. This is especially important for collision detection, path finding or adjusting the geometry of other objects (eg. vegetation, rocks) to blend with the surface of the terrain.

7 Seamless High-Quality Texturing The terrain should be textured seamlessly without any shimmering. This requires high-resolution texture data, as well as, trilinear and anisotropic filtering.

8 Multiple Views It should be possible to render the terrain multiple times using multiple views, enabling use cases like split-screen, shadow rendering, or a depth-prepass.

4.2 Evaluation of existing Literature

With these requirements established the aforementioned terrain rendering data structures and algorithms are now examined and the trade-offs fit best with these properties are discussed.

4.2.1 Implementation - CPU vs GPU

As can be observed in Figure 3.1 the most recent terrain rendering strategies have shifted more and more of their implementation to the GPU due to their highly parallel nature and drastic performance improvements compared to CPU-based methods. Especially since the introduction of compute shaders in modern graphics APIs, the LOD algorithms are mostly executed by the GPU.

4.2.2 Triangulation - Regular Grids vs TIN vs RTIN

As seen in Chapter 3 the most common terrain triangulation is the regular grid approach (see Figure 3.1). It works well with modern GPU features, such as height displacement in the vertex shader and instanced rendering. Thus the meshes for all the different parts of the terrain can be reused and batched together to save draw calls. Additionally, the actual height data can be stored alongside the remaining terrain data, unifying the implementation. Their regular structure utilizes the highly parallel nature of GPUs well. It is best used in combination with MLOD algorithms, and is especially great for further mesh refinement via tessellation shaders.

TINs are expensive to pre-compute, too rigid for highly view-dependent terrains, and hard to stitch and morph, thus they are not well suited for large-scale terrain rendering. Nevertheless, they can be ideal for less view-dependent and more static applications.

The RTIN triangulation fits especially well with the adaptive CLOD algorithms, as can be seen with the ROAM algorithm in Section 3.1 and the recently developed CBT algorithm of Section 3.8.

4.2.3 Level of Detail - Continuous vs Multi-resolutional

As previously noted terrain rendering is largely subdivided into CLOD 2.2.2 and MLOD 2.2.3 algorithms. The former offer an excellent upper bound on screen-space error while minimizing the number of triangles to render. The latter trade looser error bounds and a slightly larger geometry overhead for greatly reduced LOD calculations and better parallelization, by operating on aggregates of vertices (tiles) instead of individual triangles.

Currently, it appears no CLOD algorithm solving the second criteria exists. Particularly scaling across seven orders of magnitude is not possible with current hardware. ROAM and previous CPU-based CLOD versions do not translate well to the architectures and characteristics of modern GPU-driven renderers. Thus they are outperformed by even the simplest MLOD algorithm [10]. Although the CBT [3] algorithm produces an impressive subdivision for smaller terrains, it already consumes $128MB$ of VRAM, for a single view, at a binary tree depth of 28, covering a landscape with a modest size of $8km \times 8km$ at a resolution of $1m$. This is only a difference of four orders of magnitude or equivalently 13 powers of two, which does not reach the required seven or 25 respectively, necessary for rendering the entire earth at equal resolution. Considering that the space complexity of the algorithm scales exponentially with the tree size and linearly with the size of the terrain, it is infeasible to use it for large-scale terrain rendering.

Furthermore, culling has become an increasingly important aspect of modern GPU-driven renderers [12]. Especially, culling objects which are occluded by the terrain, or culling parts of the terrain occluded by large objects is an important performance optimization. The latter is better suited for MLOD algorithms due to their tile-based nature. With that in mind an MLOD algorithm with regular grid subdivision does fit the requirements best. Because of the rectangular nature of grids, the author chose to use a quadtree-based subdivision over a binary tree.

4.2.4 Terrain Data - Quadtree vs Clipmap

There are two prominent ways of storing terrain data: the quadtree 2.5.1 and the clipmap 2.5.2. Both offer a great hierarchical representation of the terrain data, which fits the required $O(n \cdot log(n))$ complexity. Clipmaps are easy to implement and can scale effortlessly from small regions to entire planets. [17] Their major drawback lies within requirement eight, because multiple views require the duplication of the clipmaps per view, resulting in redundant storage of large parts of the terrain in VRAM. As mentioned in 2.5.2, the texture filtering of clipmaps is not implemented in hardware, but rather has to be computed in software, which is more expensive and techniques like anisotropic filtering are unavailable entirely. Although these drawbacks are not impossible to accept, it would be better to reduce these inefficiencies.

The quadtree on the contrary is considerably better at utilizing the spherical LOD, but unfortunately, it has another problem. Trees can either be stored sparsely or densely, which means that all nodes, or only the currently required subset, have to be present. For representing the view-dependent terrain data, only the former is of any use. This poses a problem with requirement six. Contrary to a clipmap where all data of all layers has to be present and the distance to the viewer identifies the layer to access the data at, there is no simple way to look up arbitrary terrain data in a sparse quadtree in constant time. Instead, it has to be traversed top down. In Section 3.7 the "Far Cry 5" terrain renderer [12] deals with this problem by building a lookup data structure, the sector map, storing the IDs of the quadtree nodes, which can identify the best-loaded node for each position of the terrain. However, this method only works for relatively small terrains, because this lookup texture itself scales quadratically in size with regards to the terrain size and thus renders the traditional quadtree insufficient for large-scale terrain rendering.

4.3 Method Overview

As seen in the previous section, none of the aforementioned rendering methods meet all of the eight postulated requirements. For this reason, the author developed two techniques, solving separately the terrain geometry and terrain data problems. Following, the author presents a new MLOD algorithm and a novel data structure: The **Uniform Distance-Dependent Level of Detail (UDLOD)** uniformly subdivides large-scale terrains into distant-dependent regular grid tiles and the **Chunked Clipmap** efficiently shares terrain data between multiple views and can be filtered trilinearly and anisotropically. Figure 4.1 depicts a high-level overview of this novel terrain rendering method.

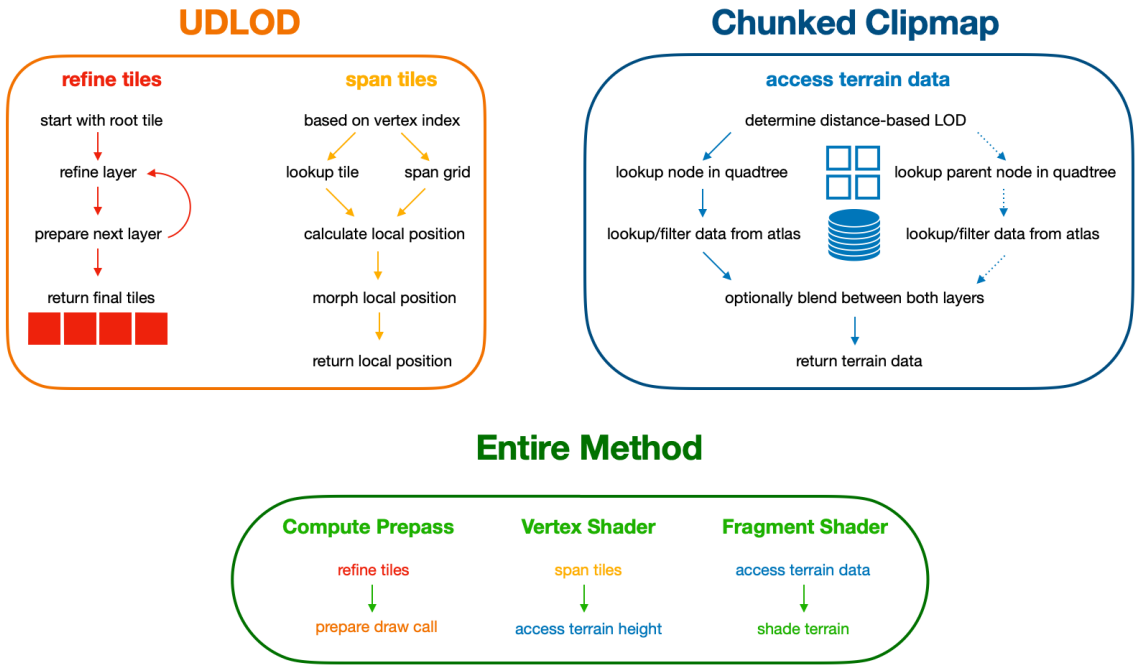


Figure 4.1: An overview of the terrain rendering method presented in this chapter. Shown on the left are the key algorithms of the UDLOD method, on the right is the access of the terrain data, and on the bottom, is the entire method, as implemented in Chapter 5.

4.4 Uniform Distance-Dependent Level of Detail (UDLOD)

For every terrain renderer, it is important to choose an appropriate and efficient geometry triangulation and LOD scheme. As discussed in Section 4.2.3, CLOD algorithms are not suitable for large-scale terrain rendering, although they offer superior geometry quality, with smaller screen-space errors compared to MLOD alternatives. Thus the author presents a modified version of the CDLOD method, called the Uniform Distance-Dependent Level of Detail (UDLOD) algorithm, which combines its continuous distant-dependent LOD with the efficient GPU quadtree subdivision of the "Far Cry 5" terrain renderer.

Its general idea is to subdivide a giant implicit quadtree covering the entire terrain during each frame on the GPU. This is achieved by traversing the tree top-down, layer by layer in a compute shader in parallel, culling tiles, and determining whether they should be subdivided or not depending on their distance to the viewer. The final tiles are then gathered in a buffer and drawn by

means of a single indirect draw call using the same vertex resolution for each of them. Additionally, they are morphed at the rim between two adjacent LOD rings, utilizing the scheme presented by the CDLOD method [11].

A major benefit of separating the geometry from the terrain data is the ability to continuously adjust the size of said tiles using the *tile_scale* parameter. This allows for a fine-grained adjustment of the quality or rather the performance characteristic of the tesselation. The next sections introduce numerous different parameters and formulas required by the UDLOD algorithm. All of them can be found in Table 1 for quick reference.

In the following some equations require the floating point modulo operation available in most programming languages, thus it is defined in Equation 4.1.

$$fmod(x, y) = x - y \cdot \left\lceil \frac{x}{y} \right\rceil \tag{4.1}$$

4.4.1 Tiling Prepass

Before the terrain can be drawn, it is subdivided into tiles, with approximately equal size in screen space, based on their distance to the viewer, during a compute shader prepass. Therefore as a first step, the terrain is covered by a single large root tile placed as the first element of a temporary tile list. This list is now refined over multiple passes, each of which corresponds to two compute shader invocations. The first one is a shader that resets the tile list indices and prepares the indirect argument buffer of the second shader, which in turn executes the tile refinement algorithm, as seen in Algorithm 1, for each tile of the current LOD layer. This algorithm decides whether or not to split a tile based on its relative distance to the viewer and either appends it to the final tile list that is drawn to the screen or splits it into its four children, which will be processed during the next pass. To write to the tile lists, indices are used, which are incremented atomically to guarantee the absence of data races.

Algorithm 1 The tile refinement algorithm, which is executed for each tile in the temporary tile list of the current layer.

```
tile ← temporary_tiles[parent_index()]
if tile is not culled then
    if tile should be subdivided then
        for child ← children of the tile do
            temporary_tiles[child_index()] ← child
        end for
    else
        final_tiles[final_index()] ← tile
    end if
end if
```

4.4.2 Tile Culling

To optimize the vertex count of the final terrain mesh the tiles can be culled in the refinement compute shader. This can be achieved by simply executing a culling algorithm on each of them before subdivision, as can be seen in Algorithm 1. The culling can use different strategies, as presented in Section 2.3. For this particular use case, discarding tiles outside the terrain and 2D horizontal frustum culling are the two easiest algorithms to implement.

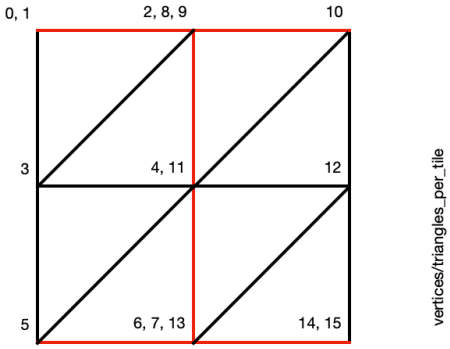
4.4.3 Drawing the Tiles

With the final tile list assembled it is now time to draw them to the frame buffer using a single indirect draw call. The following triangulation algorithm is based on the article *"Android Lesson Eight: An Introduction to Index Buffer Objects (IBOs)"* by Kevin Brothaler [23]. All vertices of the terrain mesh are determined implicitly, without the need for a vertex or index buffer.

Therefore each tile is rendered as a grid, all of which consist of the same amount of vertices. This, vertex amount can be calculated using the Equation 4.2. Thus the entire amount of vertices to draw equals the product of *vertices_per_tile* and the *tile_count*.

$$\begin{aligned} vertices\_per\_row &= 2 \cdot (grid\_size + 2) \\ vertices\_per\_tile &= vertices\_per\_row \cdot grid\_size \end{aligned} \tag{4.2}$$

To require as few vertex shader invocations as possible the **triangle strip** topology is used when drawing the grids. Each column can be easily represented by such a strip, but special care has to be taken when connecting them together to prevent drawing malformed geometry. By inserting the first and last index of each column twice, four degenerate triangles will form, which reset the vertex list back to a valid state and bridge the gap between both columns, without drawing any geometry. This also works for connecting successive tiles with arbitrary positions together, which means that the order of the tiles in the tile list can be arbitrary as well. As can be seen in Figure 4.2 this triangulation results in a vertex-to-triangle ratio converging towards one.



grid_size	2	4	6	8	10	12	14	16
vertices_per_tile	16	48	96	160	240	336	448	576
triangles_per_tile	8	32	72	128	200	288	392	512

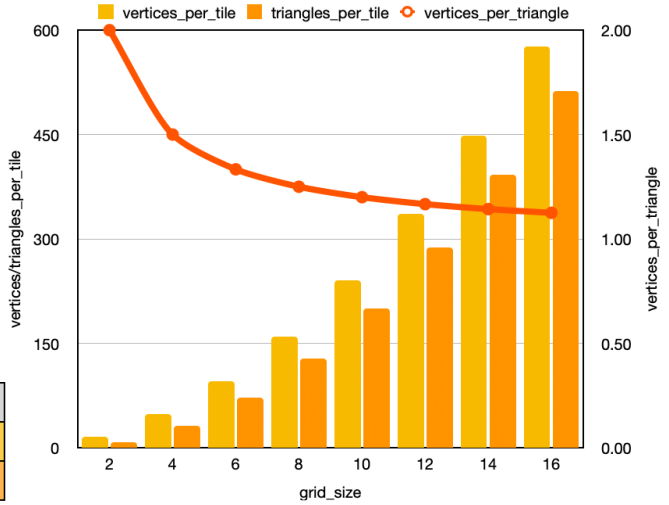


Figure 4.2: Shows a grid of size two on the left, with the degenerate triangles colored in red, as well as an evaluation of the triangle-to-vertex efficiency of this triangulation method for different grid sizes.

To span this grid, first the *tile_index* and *grid_index* of the vertices are calculated according to Equation 4.3. The former is responsible for retrieving the *tile_coords* and *tile_size* from the final tile list. While the latter is used to build the grid, as seen in Figure 4.2, using the three equations found in Equation 4.4. Subsequently, it has to be scaled and translated according to the tile information, using the Equation 4.5, to fit seamlessly into the terrain's surface.

$$\begin{aligned} tile\_index &= vertex\_index \div vertices\_per\_tile \\ grid\_index &= fmod(vertex\_index, vertices\_per\_tile) \end{aligned} \tag{4.3}$$

$$\begin{aligned} row\_index &= clamp(fmod(grid\_index, vertices\_per\_row), 1, vertices\_per\_row - 2) - 1 \\ column\_index &= grid\_index \div vertices\_per\_row \\ grid\_position &= \begin{bmatrix} column\_index + fmod(row\_index, 2) \\ row\_index \div 2 \end{bmatrix} \end{aligned} \tag{4.4}$$

$$local\_position = (tile\_coords + \frac{grid\_position}{grid\_size}) \cdot tile\_size \cdot tile\_scale \tag{4.5}$$

4.4.4 Tile Morphing

Now that all grids are positioned at their appropriate locations, it is important to morph adjacent tiles of different LOD and size seamlessly together, to avoid T-Junctions. This can be achieved similarly to the method proposed by Filip Strugar on a per-vertex basis. [11] For each vertex, the *morph* factor is determined using the approximate distance between it and the viewer, described in Section 4.5.2. This value is then used to shift every other vertex towards its neighbor using Equation 4.6.

$$\begin{aligned} even\_grid\_position &= 2 \cdot \left\lfloor \frac{grid\_position}{2} \right\rfloor \\ morphed\_local\_position &= local\_position - \\ &\qquad morph \cdot \frac{even\_grid\_position}{grid\_size} \cdot tile\_size \cdot tile\_scale \end{aligned} \tag{4.6}$$

Finally, the height value has to be sampled from the chunked clipmap or any other terrain data representation used. Then the vertex can be vertically displaced by this amount and transformed into the world space.

4.5 Chunked Clipmap

With the terrain geometry taken care of, it is important to choose an efficient terrain data representation. As seen in Section 4.2.4 both quadtrees and clipmaps do pose some major drawbacks. That is why the author proposes a novel data structure, combining both of their strengths: the chunked clipmap. The basic idea of this approach is to represent all terrain data as a quadtree where each layer is clipped to a maximum size similar to a clipmap. A comprehensive list of all parameters and variables alongside a short description can be found in Table 2.

4.5.1 Structure of a Chunked Clipmap

The chunked clipmap consists of two separate parts, which can be combined to access the best available terrain data, for a specific view, at any position. They are called the **Node Atlas** and the **Clipped Quadtree**. Both are illustrated in Figure 4.3.

The **Node Atlas** is a container that stores all currently loaded terrain data nodes and assigns a unique handle, also called the *atlas_index*, to each of them. It can store multiple different types of terrain data per node, provided that they are loaded simultaneously.

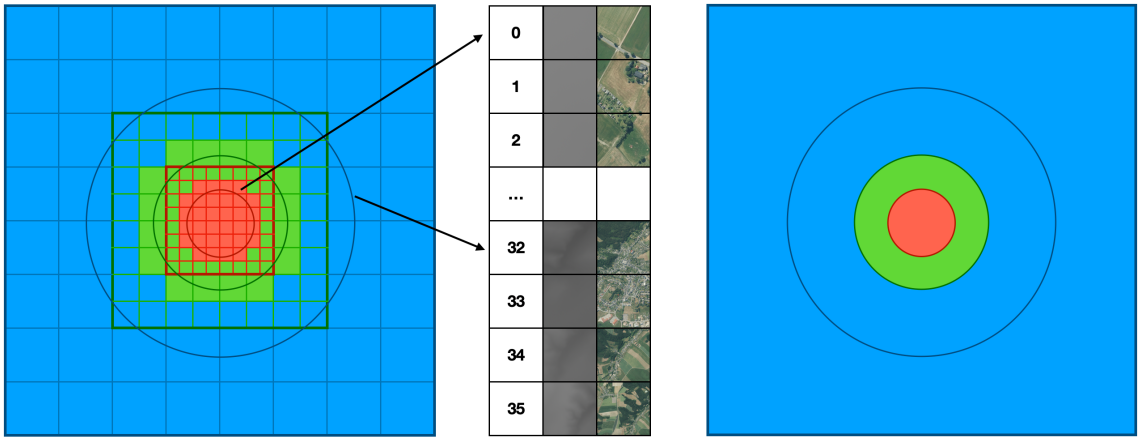


Figure 4.3: Depicts the clipped quadtree (left) next to the node atlas (middle) and the resulting LOD distribution (right). The quadtree entries store an *atlas_index* and the *atlas_lod*.

The **Clipped Quadtree** is a small lookup data structure responsible for mapping world positions to coordinates inside the corresponding node atlas. It is represented by a three-dimensional matrix of size $node\_count \times node\_count \times lod\_count$, where each entry stores the *atlas_index* and the *atlas_lod* of the best available node spanning across the corresponding area, which can be seen in Figure 4.4. Each layer covers increasingly large portions of the terrain, which are centered under the viewer, similar to clipmaps, and uses toroidal addressing, to stay consistent while roaming over the terrain, as well.

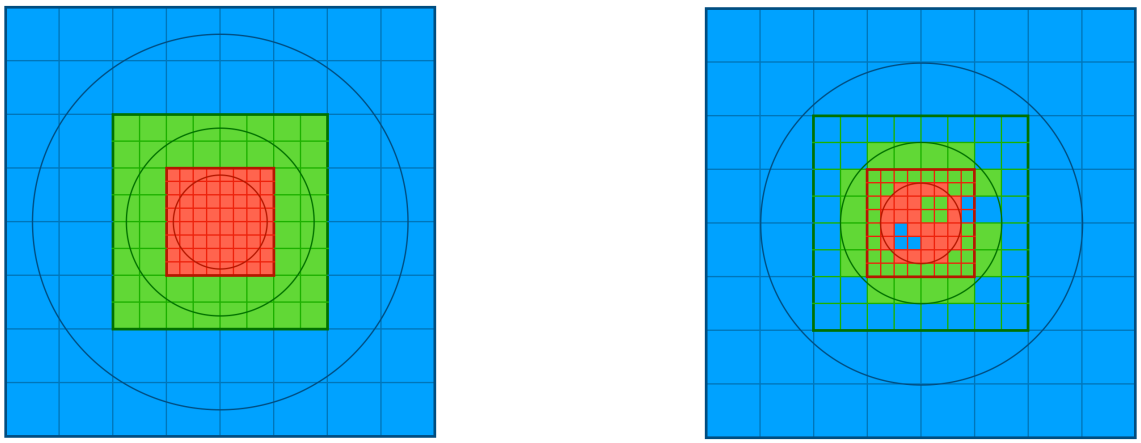


Figure 4.4: Showcases two different states of the quadtree. On the left, the quadtree is fully loaded whereas on the right, some nodes are not, thus the quadtree falls back to lower-resolution nodes of the layers above.

Both can be implemented using three-dimensional arrays on the CPU and array textures on the GPU. Each view possesses a unique clipped quadtree, but all of them share the same node atlas, so no terrain data has to be duplicated.

4.5.2 Approximatig the Distance to the Viewer

To access and update the chunked clipmap the distance-dependent LOD has to be provided. It is calculated from the three-dimensional distance between the point on the terrain and the position of the viewer. The horizontal coordinate can be determined trivially, but the vertical displacement of the terrain poses a major problem. To obtain the height value at any point on the terrain's surface, it has to be sampled from the chunked clipmap. This cyclic dependency can only be resolved by sampling multiple times with increasing accuracy, or by simply approximating the height value used for the LOD selection.

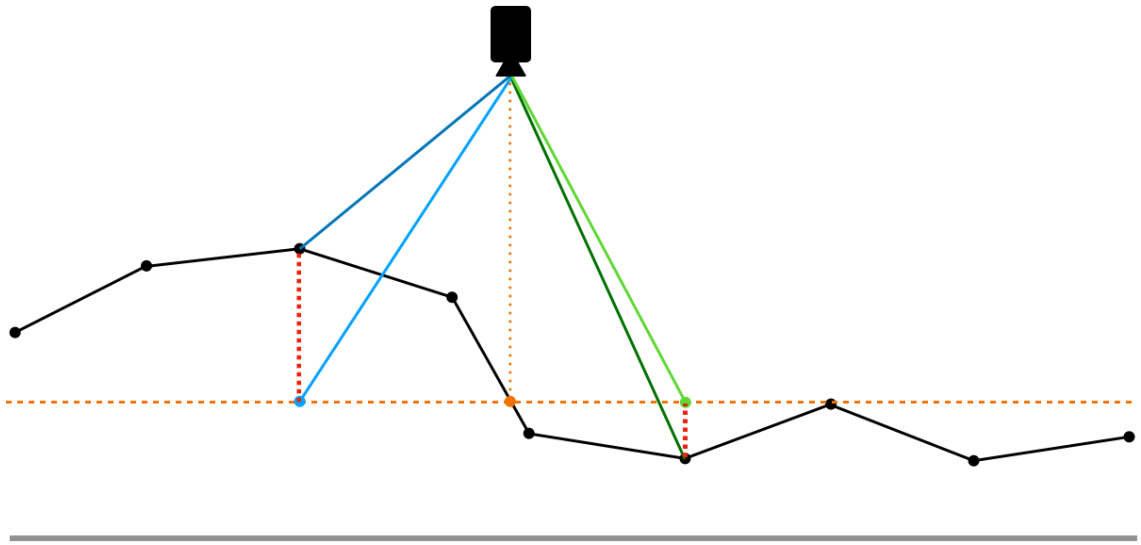


Figure 4.5: Illustrates the cross-section of the terrain. The horizontal orange line represents the approximated height. The green and blue lines depict the true and approximated distances for two points of the terrain. The red lines indicate the vertical error introduced by these approximations.

One such approximation can be achieved by using the height of the terrain under the viewer as the height value of all distance calculations. The author found that this approximation, which is illustrated by Figure 4.5, resulted in no visible artifacts. The approximate distance can then be calculated using Equation 4.7, where x and y denote the horizontal position of the point of interest in world space.

$$distance(x,y) = \left\| \begin{bmatrix} x \\ approximate\_height \\ y \end{bmatrix} - view\_position \right\| \tag{4.7}$$

4.5.3 Accessing the Terrain Data

The side length of each node is determined by the *node_size* value, which is calculated using Equation 4.8. This formula scales the *leaf_node_size* parameter, which denotes the size of the smallest nodes in local space, by the exponential increase in size caused by the nature of the quadtree.

$$node\_size(lod) = leaf\_node\_size \cdot 2^{lod} \tag{4.8}$$

Additionally, there is a customizable *view_distance* parameter, that defines the radius of a sphere around the viewer. It is specified in multiples of the *leaf_node_size* and constrains the area covered by each LOD, to allow for fast and easy three-dimensional distance-dependent data access. This parameter must be smaller than $(node\_count - 1)$. Otherwise the quadtree may be looked up outside its bounds, resulting in the retrival of invalid data.

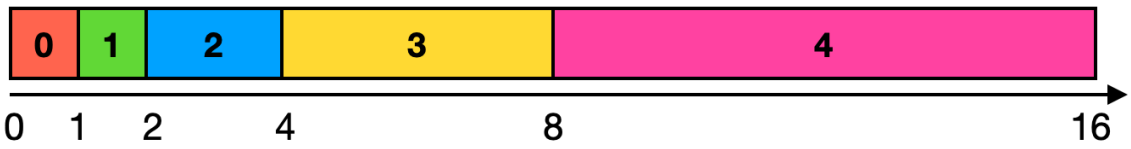


Figure 4.6: Depicts the LOD distribution (colored band) depending on the distance to the viewer (bottom axis), which is measured in multiples of the *view_distance*.

To retrieve the appropriate *atlas_index* and the *atlas_coords* identifying the location of the data inside the accessed node, some simple arithmetic has to be performed. At first, the LOD layer in the quadtree should be determined. Therefore the LOD distribution depicted in Figure 4.6 is used. It makes sure that the resolution is indirectly proportional to the approximate distance between the position and the viewer in world space. This can be achieved by using Equation 4.9, which additionally clamps the LOD between zero and the maximum level.

$$quadtree\_lod(distance) = clamp(\log_2(2 \cdot \frac{distance \cdot leaf\_node\_size}{view\_distance}), 0, lod\_count - 1) \tag{4.9}$$

With the LOD evaluated, the next step is computing the quadtree coordinates, identifying the node at the requested position. This is done by Equation 4.10, using toroidal addressing. Then the *atlas_index* and *atlas_lod* can be looked up in the clipped quadtree data structure.

$$quadtree\_coords(position, quadtree\_lod) = \left\lfloor fmod(\frac{position}{node\_size(quadtree\_lod)}, node\_count) \right\rfloor \tag{4.10}$$

Finally, the correct coordinates, inside the node, have to be computed, as can be seen in Equation 4.11. To compute them the just determind *atlas_lod* is used. This makes sure that even if the node of the proper *quadtree_lod* is not loaded yet, the quadtree can fall back to a large node of lower resolution. An example of this can be seen on the right of Figure 4.4. With both the *atlas_index* and the *atlas_coords* available it is now possible to access any terrain data inside the node atlas at the requested position.

$$atlas\_coords(position, atlas\_lod) = fmod(\frac{position}{node\_size(atlas\_lod)}, 1) \tag{4.11}$$

4.5.4 Texture Filtering for Chunked Clipmaps

Texture filtering of compound textures is a difficult problem to solve. The hardware-accelerated texture filtering units of modern GPUs, accessible by means of a **texture sampler**, are limited to operating on a single texture (layer) at a time. This creates a problem when sampling near the edges of a node because information about adjacent texels resides in a different texture or texture layer. The same problem exists for the mipmaps, even though the chunked clipmap (or even the standard clipmap) stores the down-sampled texture data in a node on another layer, it is not accessible for filtering.

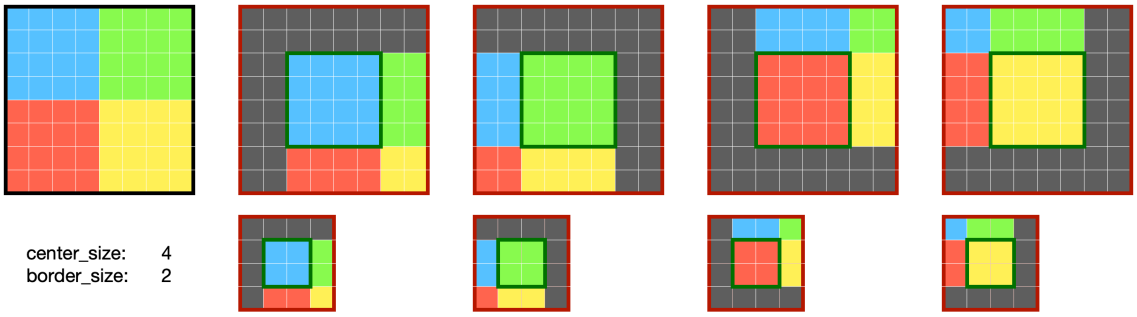


Figure 4.7: Illustrates the four nodes required to store the texture on the left, using a *center_size* of 4 and a *border_size* of 2, together with their first mipmap.

A solution to this problem is storing data redundantly. The problem of the missing neighboring pixel information can be solved by surrounding each node with a border of overlapping pixel data. Similarly, the mipmap data can be duplicated per node as well. This slight overhead enables seamless trilinear and even anisotropic texture filtering. Figure 4.7 shows an example of such a partitioning, with a *center_size* of 4 and a *border_size* of 2.

$$adjusted\_atlas\_coords(atlas\_coords) = \frac{atlas\_coords \cdot center\_size + border\_size}{center\_size + 2 \cdot border\_size} \tag{4.12}$$

Because of the additional border, the *atlas_coords* will have to be adjusted. Therefore the atlas coordinates have to be remapped to fit the center of the node, which is achieved by Equation 4.12.

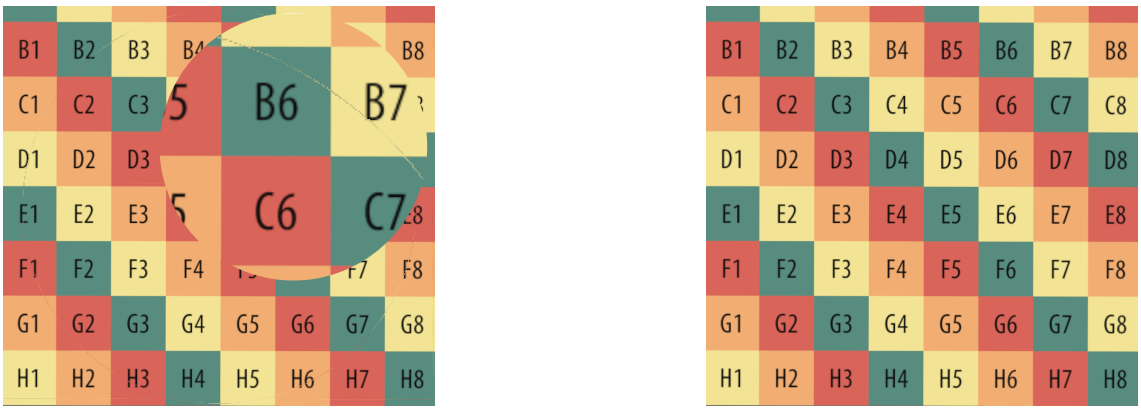


Figure 4.8: Shows a comparison between sampling using implicit gradients on the left and explicit ones on the right. Notice the fine line of pixel errors at the LOD boundary on the left.

Unfortunately adjusting *atlas_coords* is not enough to get trilinear and anisotropic filtering fully working. As seen on the left in Figure 4.8, there is still a visual error at the edge where adjacent textures meet, when relying on the hardware's implicit derivate calculation. This error is because the *atlas_coords* variable is not continuous in those regions. To prevent this artifact, the texture sampling gradients have to be calculated explicitly in the fragment shader. This can be achieved using the Equation 4.13, which scales the derivative of the horizontal position of the fragment on the terrain according to the underlying node's dimensions. Using these gradients together with the explicit variant of the texture filtering function, these artifacts can be entirely mitigated, as shown on the right in Figure 4.8.

$$\begin{aligned} adjusted\_ddx(position, atlas\_lod) &= \frac{ddx(position)}{center\_size \cdot 2^{atlas\_lod}} \\ adjusted\_ddy(position, atlas\_lod) &= \frac{ddy(position)}{center\_size \cdot 2^{atlas\_lod}} \end{aligned} \tag{4.13}$$

4.5.5 Layer Blending

Complicated texture filtering is not the only issue caused by the multi-resolution representation of the terrain data. Another problem is caused by a harsh jump in resolution at the transition between adjacent LODs. Although the spherical LOD greatly reduces its effect down to a line, instead of entire nodes popping in and out of the view, there still is a noticeable discontinuity visible on the screen.

To solve this issue, we sample the two LOD layers at the fringe of the LOD rings and interpolate between them. Depending on the range in which the data is blended, this will result in an imperceptible and smooth transition. Figure 4.9 depicts such a blended LOD distribution.

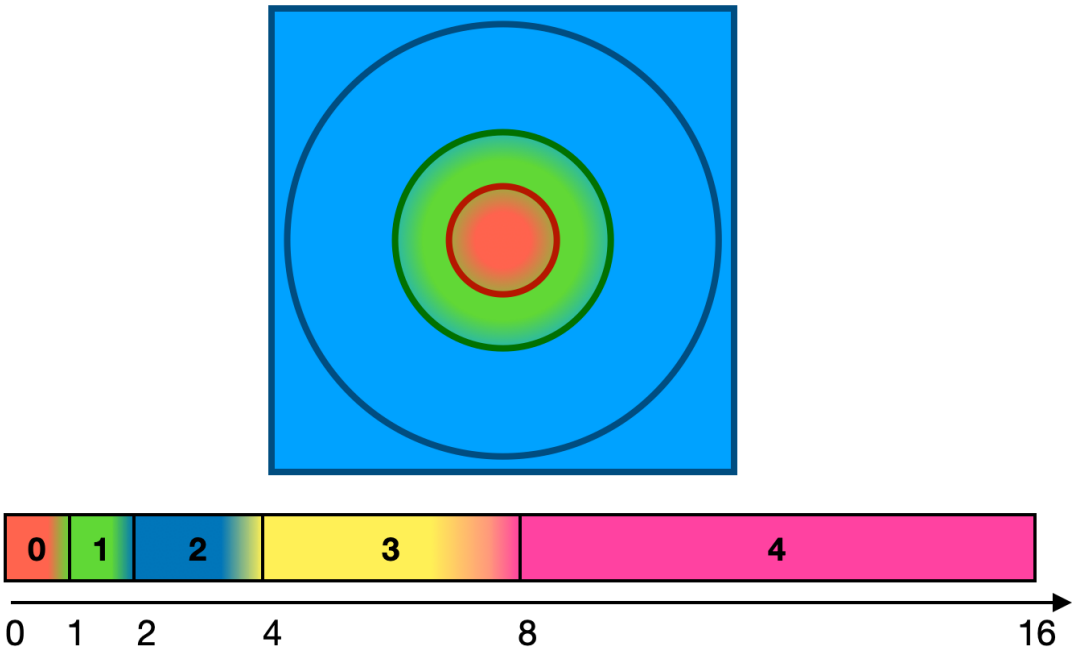


Figure 4.9: Depicts the blended LOD distribution in one and two dimensions. The distance (bottom axis) is measured in multiples of the *view_distance*.

4.5.6 Node Preprocessing

To partition the nodes from the source data, three preprocessing steps are required, as can be seen in Figure 4.10. At first, we split into the smallest LOD nodes one or multiple source images storing the terrain data in a nonoverlapping way, according to the desired center and border size. During this step, the overlapping border data can be simply copied for each node as well. With all of the level zero nodes available, they can now be downsampled into the next LOD layer. Therefore the center of four adjacent nodes is combined and downscaled linearly. The third step consists of stitching adjacent nodes together, by copying over the border data from their neighbors. Borders without a neighbor can fill the border by extending their center along the edges. These last two steps are now repeated for each additional LOD layer until the entire node pyramid is built.

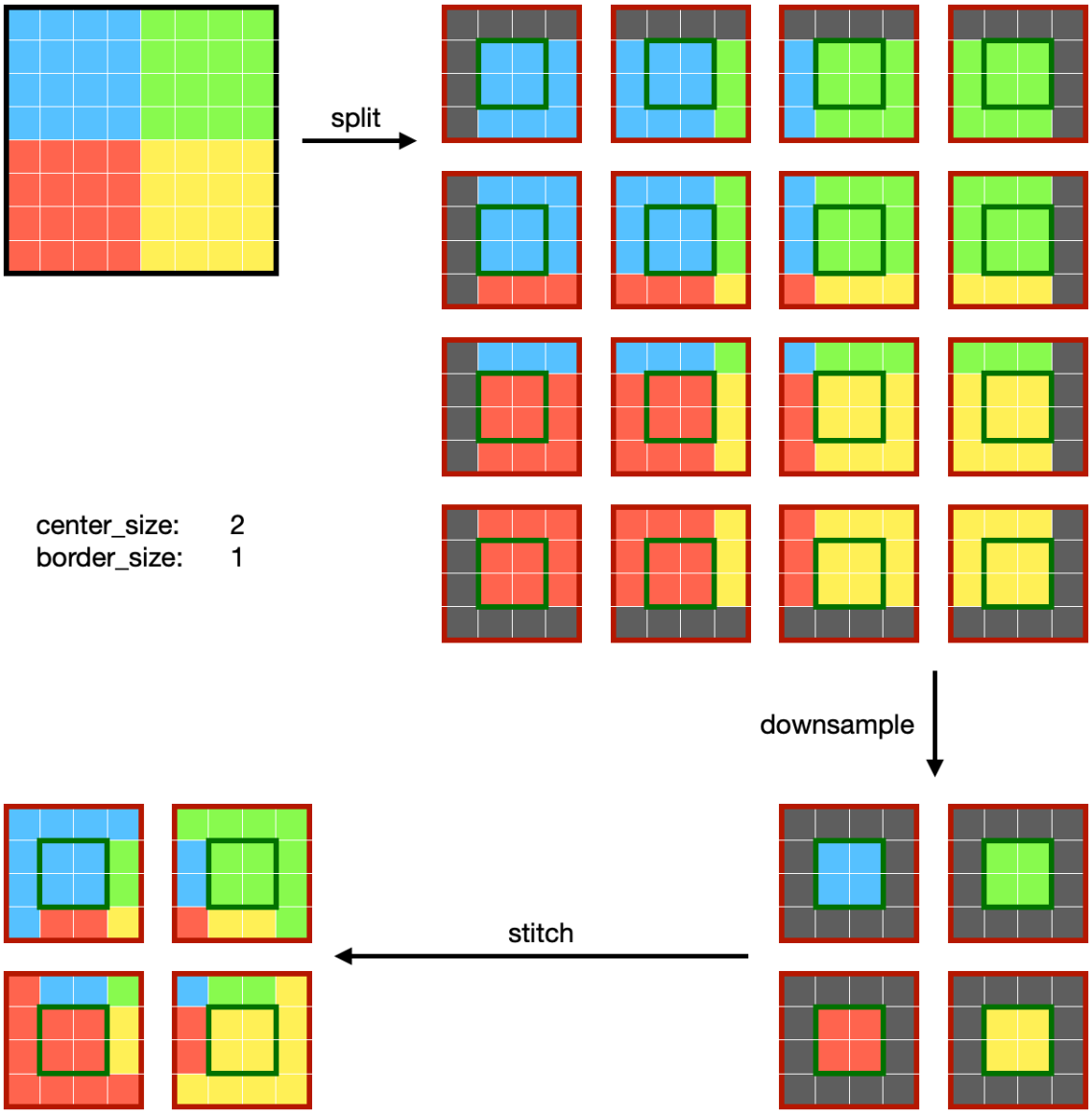


Figure 4.10: Illustrates the three steps (split, downsample and stitch) required to preprocess the source terrain data into the node pyramid.

4.5.7 Updating the Chunked Clipmap

Because chunked clipmaps only cache a view-dependent subset of all terrain data, it is important to keep this information up to date whenever the viewer's position changes. This update is performed by centering the clipped quadtree under the viewer while aligning the position of each layer to the nearest multiple of the corresponding *node_size*. Then for each entry of the quadtree, the distance to the viewer is compared to the *load_distance* parameter scaled by the LOD of the entry, thereby it is decided whether the node at that position should be loaded. All updates, of each clipped quadtree of the terrain, are gathered and then compared against the node atlas. Nodes with differing states in the update and the node atlas are then loaded depending on whether they are requested by at least one view or unloaded if they are not requested by any of the views.

Whenever a node has finished loading it is copied into the node atlas and marked as available. Due to the asynchronous nature of the node loading operation, the quadtrees can not assume that their requested state is equal to the current state of the node atlas. Thus in a second adjustment step, each entry of the clipped quadtree is updated with the best currently loaded node available in the node atlas.

This four-step process is summarized by Figure 4.11. Nodes that are loaded in the atlas, but are no longer in use may also be cached as long as the slot is not required by a newly loaded node.

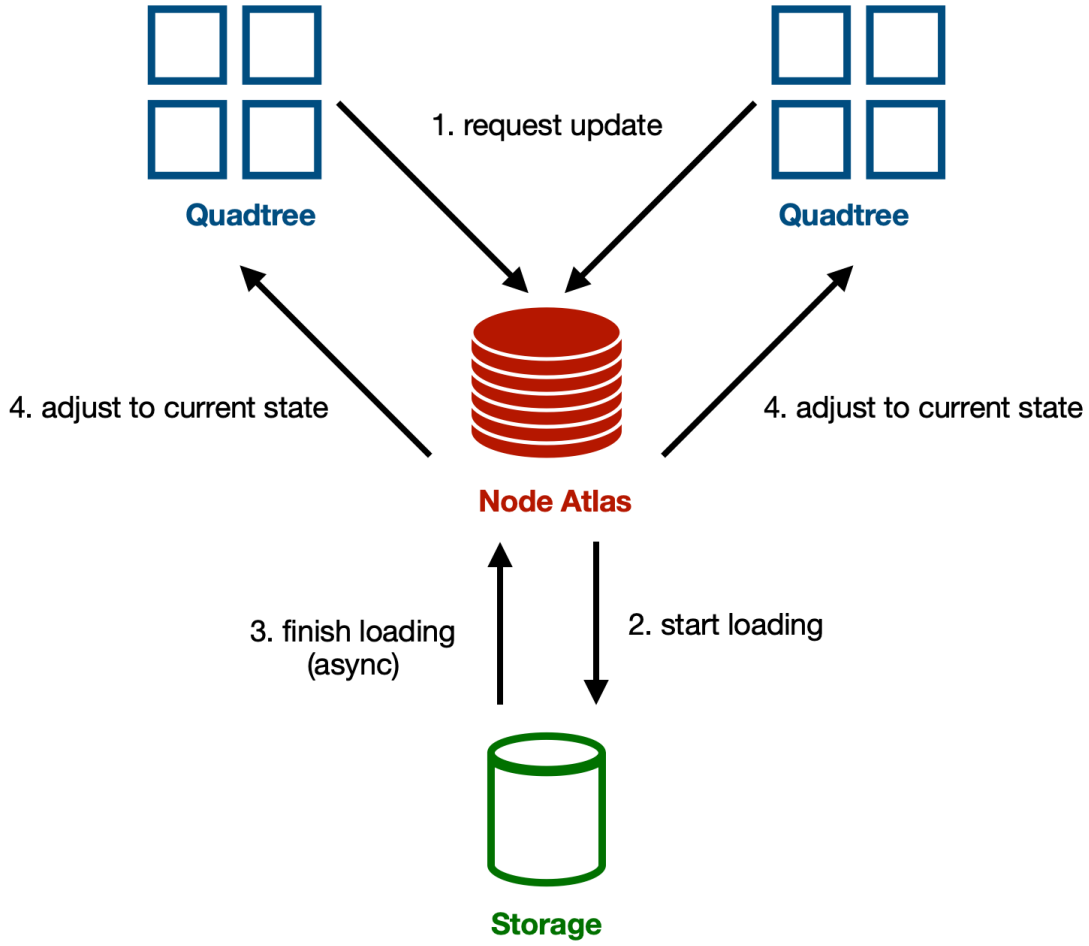


Figure 4.11: Shows the four steps required for updating a chunked clipmap. They are performed consecutively during each frame.

Chapter 5

Implementation

The following chapter is dedicated to outlining the challenges and important details encountered during the implementation of the previously introduced terrain rendering method. The application (https://github.com/kurtkuehnert/terrain_renderer), which was written alongside this thesis, is developed using the programming language Rust and the upcoming graphics API WebGPU. This choice was made to try out new technology in the computer graphics domain, which promises some great benefits over the traditional C++ and OpenGL (or Direct X) alternatives.

Rust is a modern general-purpose programming language that focuses on performance, reliability, and productivity. It offers strong memory safety and concurrency guarantees, which make multi-threading easy to do, as well as preventing the occurrences of segmentation faults and data races.

WebGPU is a modern cross-platform graphics API, which provides 3D-rendering and GPU computing, to web browsers and native applications alike. Unlike its predecessor WebGL, it introduces many new capabilities, like GPU compute, storage buffers, and indirect draw commands to the Web, which are utilized by the implementation of this terrain renderer. Additionally, WebGPU comes with its own shading language called WGSL, which is used by this implementation as well as all shader snippets of this thesis. It is a great choice for research implementations because it is internally based on the existing platform native APIs: Vulkan, Metal, and Direct3D 12, which means that every program developed using WebGPU can by definition also be implemented using one of the other three APIs. At the time of writing the specification of WebGPU is not yet fully finalized and thus is not yet enabled by default in browsers today, but libraries implementing the current draft of the specification already exist. Such libraries include Dawn (C++) and wgpu (Rust, used by this project).

Bevy To alleviate the burden of having to implement everything from scratch the author chose to utilize the low-level open-source game engine framework called Bevy, which is responsible for handling the data loading, organizing the different components, managing the window, input, and graphics contexts, and determining the general structure of the entire application. It provides all of this functionality as an easy-to-use cross-platform library. Additionally, it offers the benefit of integrating with other plugins to combine projects for more extensive, detailed, and interesting scenes.

5.1 Terrain Geometry

In the following sections, the implementation of the UDLOD algorithm will be discussed. The code snippets, which are written in WGSL, rely on types, functions, uniform values, and shader bindings that can be seen in Figure 1.

5.1.1 Tile Refinement

Implementing tile refinement, as described in Section 4.4.1, comes with the inherent challenge of parallelizing the algorithm on the GPU. All parallel executions of the prepass read and write from the same temporary tile list and must output tiles into the final tile list. To prevent duplicating, or skipping work, it is important to never access these buffers simultaneously at the same index. This can be achieved by incrementing them atomically, as can be seen in Figure 5.1.

```
fn parent_index(id: u32) -> i32 {
    return i32(MAX_TILE_COUNT - 1u) * clamp(parameters.counter, 0, 1)
           - i32(id) * parameters.counter;
}

fn child_index() -> i32 {
    return atomicAdd(&parameters.child_index, parameters.counter);
}

fn final_index() -> i32 { return atomicAdd(&parameters.final_index, 1); }
```

Figure 5.1: These three functions are responsible for retrieving the indices used to access the tiles during refinement. They are updated atomically to guarantee mutual exclusive access.

Unlike the approach used by the "Far Cry 5" terrain renderer [12], this implementation only uses a single temporary tile list, which is indexed from the front and the back simultaneously. One index references the parent tiles, and the other identifies the position of newly written child tiles, inside the buffer. After each layer has been processed, the indices are swapped and the no longer required parent tiles will be overwritten by the child tiles of the following refinement.

```
@compute @workgroup_size(1, 1, 1)
fn prepare_next() {
    if (parameters.counter == 1) {
        parameters.tile_count =
            u32(atomicExchange(&parameters.child_index, i32(MAX_TILE_COUNT - 1u)));
    }
    else {
        parameters.tile_count =
            MAX_TILE_COUNT - 1u - u32(atomicExchange(&parameters.child_index, 0));
    }

    indirect_buffer.workgroup_count.x = (parameters.tile_count + 63u) / 64u;
    parameters.counter = -parameters.counter;
}
```

Figure 5.2: This shader is invoked after each pass of tile refinement to prepare the indices, the counter, and the tile_count for the next one.

The shader shown in Figure 5.2 is executed in between successive executions of the refinement shader and inverts the counter, determining the increment direction of the parent and child indices. Additionally, the indirect buffer, specifying the number of parent tiles to refine next, is updated.

5.1.2 Tile Frustum Culling

As mentioned in Section 4.4.2, the number of tiles and by extension vertices to render can be decreased drastically by utilizing basic frustum culling. Therefore the implementation uses an optimized frustum culling algorithm, which is part of the tile refinement compute shader on the GPU. The six planes, defined by their normal and their distance to the origin, are extracted from the view frustum by the CPU and transmitted to the GPU as part of a uniform buffer. The shader of Figure 5.3 shows the entire algorithm as it is defined in the blog post "*Geometric Approach – Testing Boxes II*". [24]

```
fn frustum_cull(tile: Tile) -> bool {
    let size = f32(tile.size) * TILE_SCALE;
    let aabb_min = vec3<f32>(f32(tile.coords.x), 0.0, f32(tile.coords.y)) * size;
    let aabb_max = vec3<f32>(aabb_min.x + size, MAX_HEIGHT, aabb_min.z + size);

    for (var i = 0; i < 6; i = i + 1) {
        let plane = FRUSTUM_PLANES[i];
        var p_corner = vec4<f32>(aabb_min, 1.0);
        var n_corner = vec4<f32>(aabb_max, 1.0);
        if (plane.x >= 0.0) { p_corner.x = aabb_max.x; n_corner.x = aabb_min.x; }
        if (plane.y >= 0.0) { p_corner.y = aabb_max.y; n_corner.y = aabb_min.y; }
        if (plane.z >= 0.0) { p_corner.z = aabb_max.z; n_corner.z = aabb_min.z; }
        if (dot(plane, p_corner) < 0.0) { return true; }
        else if (dot(plane, n_corner) < 0.0) { return false; }
    }
    return false;
}
```

Figure 5.3: The frustum culling function returns true if the tile can be culled, or false otherwise.

To decide whether or not a tile is visible, for each plane the closest and furthest vertex along its normal is determined. If the closest vertex is located on the outside (relative to the frustum) of the plane, then it can be culled safely because no other vertex can lie within, otherwise, if the most distant vertex in relation to the normal lies on the inside (relative to the frustum) of the plane, then the box must intersect the plane. Should the tile pass all tests without being found to intersect or lie outside the frustum, then the tile has to be rendered and can not be culled. [24]

5.1.3 UDLOD Vertex Shader

The vertex shader of the UDLOD algorithm is responsible for mapping each vertex index to a horizontal local position. Afterwards, the steps outlined in Section 4.4.3 are necessary and their implementation can be seen in Figure 5.4. At first, the shader computes the tile index and the grid index. Then the tile data is fetched from the tile list and the grid position is calculated using the topmost function of Figure 5.5. Finally, the grid position is scaled, translated, and morphed at the rim between two LOD rings, according to the tile information.

```
let tile_index = vertex.index / VERTICES_PER_TILE;
let grid_index = vertex.index % VERTICES_PER_TILE;

let tile = tiles.data[tile_index];
let grid_position = calculate_grid_position(grid_index);

let local_position = calculate_local_position(tile, grid_position);
```

Figure 5.4: Shows the vertex shader section of the UDLOD algorithm.

This last step proved to be the most complicated. The translated local position is computed and the corresponding world position is estimated using the approximate height. Afterward, the morph value is calculated, using the middle function of Figure 5.5. Finally, the local position of every other vertex is interpolated towards its neighbor, based on this ratio.

```
fn calculate_grid_position(grid_index: u32) -> vec2<u32>{
    let row_index =
        clamp(grid_index % VERTICES_PER_ROW, 1u, VERTICES_PER_ROW - 2u) - 1u;
    let column_index = grid_index / VERTICES_PER_ROW;
    return vec2<u32>(column_index + (row_index & 1u), row_index >> 1u);
}

fn calculate_morph(tile_size: u32, world_position: vec4<f32>) -> f32 {
    let viewer_distance = distance(world_position.xyz, VIEW_POSITION.xyz);
    let morph_distance = MORPH_DISTANCE * f32(tile_size << 1u);
    return clamp(1.0 - (1.0 - viewer_distance / morph_distance)
           / MORPH_RANGE, 0.0, 1.0);
}

fn calculate_local_position(tile: Tile, grid_position: vec2<u32>) -> vec2<f32> {
    let size = f32(tile.size) * TILE_SCALE;
    let local_position =
        (vec2<f32>(tile.coords) + vec2<f32>(grid_position) / GRID_SIZE) * size;

    let world_position = approximate_world_position(local_position);
    let morph = calculate_morph(tile.size, world_position);
    let even_grid_position = vec2<f32>(grid_position & vec2<u32>(1u));

    return local_position - morph * even_grid_position / GRID_SIZE * size;
}
```

Figure 5.5: Depicts the local position calculation of the UDLOD algorithm. The topmost function maps the grid position to each vertex based on the grid index. The middle function determines the ratio at which to morph vertices with even grid positions.

5.2 Terrain Data

Three steps must be performed to access the terrain data. At first, the LOD, at which the information is required, and the blend ratio used to interpolate between two adjacent layers, is determined. Then the specific node storing the best available data is identified for one or both of them. Finally, the desired information is sampled and filtered.

5.2.1 Multiple Terrain Attachments

For an extensible implementation of a terrain renderer, it is crucial to represent multiple different kinds of terrain data with varying resolutions. Therefore the chunked clipmaps, outlined in Section 4.5, have to be adapted to support an arbitrary amount of different terrain data layers, so-called **terrain attachments**.

Each attachment possesses a unique atlas texture array as part of the node atlas, which stores all of its currently loaded data. For each node, one associated texture for each attachment is stored, which all have to finish loading before the information can be accessed.

The function shown in Figure 5.6 combines all the coordinate remapping, described in Section 4.5.3, required for accessing the node at a horizontal position and a specific LOD.

```
fn lookup_node(lod: u32, local_position: vec2<f32>) -> NodeLookup {
    let quadtree_lod = min(lod, LOD_COUNT - 1u);
    let quadtree_coords = vec2<i32>((local_position / node_size(quadtree_lod))
                          % f32(NODE_COUNT));
    let lookup = textureLoad(quadtree, quadtree_coords, i32(quadtree_lod), 0);
    let atlas_index = i32(lookup.x);
    let atlas_lod = lookup.y;
    let atlas_coords = (local_position / node_size(atlas_lod)) % 1.0;
    return NodeLookup(atlas_lod, atlas_index, atlas_coords);
}
```

Figure 5.6: This function is used to look up the node, at the given position and LOD, in the quadtree of this viewer.

5.2.2 Blending

Before the data can be sampled from a node, at first the blend ratio and LOD are computed. Similar to morphing, the blend ratio is calculated based on the world position of the point and its distance to the viewer. For that, the function depicted in Figure 5.7 is used.

```
fn calculate_blend(world_position: vec4<f32>) -> Blend {
    let viewer_distance = distance(world_position.xyz, VIEW_POSITION.xyz);
    let log_distance = max(log2(2.0 * viewer_distance / BLEND_DISTANCE), 0.0);
    let ratio = (1.0 - log_distance % 1.0) / BLEND_RANGE;
    return Blend(u32(log_distance), ratio);
}
```

Figure 5.7: This function computes the blend ratio and the LOD used to sample the terrain data.

The following two code snippets, shown in Figure 5.8, describe the entire process of gathering the terrain data in the vertex and fragment shader respectively. The world position is approximated in the vertex shader once more, the blend value is calculated, and the node corresponding to the position is looked up in the quadtree. If the blend ratio lies within the range of zero and one, then the terrain data is sampled a second time, using the lower LOD of the parent node, and both sampled values are blended together. To use trilinear and anisotropic filtering in the fragment shader, the gradients of the local position in the x and y directions are determined and passed to the terrain data sampling function.

```
let world_position = approximate_world_position(local_position);
let blend = calculate_blend(world_position);
let lookup = lookup_node(blend.lod, local_position);
var height = sample_height(lookup);

if (blend.ratio < 1.0) {
    let lookup2 = lookup_node(blend.lod + 1u, local_position);
    let height2 = sample_height(lookup2);
    height      = mix(height2, height, blend.ratio);
}
```

```
let ddx    = dpdx(local_position);
let ddy    = dpdy(local_position);
let blend = calculate_blend(world_position);
let lookup = lookup_node(blend.lod, local_position);
var data   = sample_fragment_data(lookup, ddx, ddy);

if (blend.ratio < 1.0) {
    let lookup2 = lookup_node(blend.lod + 1u, local_position);
    let data2   = sample_fragment_data(lookup2, ddx, ddy);
    data        = blend_fragment_data(data, data2, blend.ratio);
}
```

Figure 5.8: Shows the vertex (top) and the fragment (bottom) shader section, that is used to access the data from a chunked clipmap.

5.2.3 Sampling and Filtering

As seen in the previous Section 4.5, the sampling and filtering of chunked clipmaps do require special care. Additionally, the previously computed atlas coordinates have to be adjusted to fit the center of the node, which actually contains the non-overlapping node data. This can be implemented using a single multiplication and addition, as can be seen in Figure 5.9. To fetch the best available data from the node (e.g. in the vertex shader) a simple texture sample at mip level zero is used.

```
// for each attachment adjust the coordinates and sample its data
let attachment_coords = lookup.atlas_coords * ATTACHMENT_SCALE + ATTACHMENT_OFFSET;
let data = textureSampleLevel(attachment_atlas, atlas_sampler, attachment_coords,
                              attachment_index, 0.0);
```

Figure 5.9: Depicts how to sample data from an attachment using bilinear filtering.

The derivatives of the trilinear and anisotropic filters used in the fragment shader should also be adjusted. They have to be divided by the node size and scaled according to the individual attachment size. At last, the data can be retrieved using the texture sample gradient function, as seen in Figure 5.10.

```
let ddx = ddx / f32(1u << lookup.atlas_lod);
let ddy = ddy / f32(1u << lookup.atlas_lod);

// for all attachments adjust the coordinates and derivatives
let attachment_coords = lookup.atlas_coords * ATTACHMENT_SCALE + ATTACHMENT_OFFSET;
let attachment_ddx = ddx / ATTACHMENT_SIZE;
let attachment_ddy = ddy / ATTACHMENT_SIZE;

// sample the data of the attachment
let data = textureSampleGrad(attachment_atlas, atlas_sampler, attachment_coords,
                             attachment_index, attachment_ddx, attachment_ddy);
```

Figure 5.10: Depicts how to sample data from an attachment using trilinear/anisotropic filtering.

5.2.4 Normal Calculation

Although the normals could be stored as yet another terrain attachment, they can also be computed from the height data in the fragment shader. This offers many benefits, such as reduced VRAM usage and simpler updating of the terrain data. To calculate these normals the function seen in Figure 5.11 is used. Therefore the height map is sampled four times with an offset of one texel in the x- or z-axis. These are then used to span a vector perpendicular to the terrain's surface, the approximate terrain normal. This calculation also uses the texture sample gradient function to enable trilinear filtering as well.

```
fn calculate_normal(coords: vec2<f32>, atlas_index: i32, atlas_lod: u32,
                    ddx: vec2<f32>, ddy: vec2<f32>) -> vec3<f32> {
    let offset = 1.0 / HEIGHT_ATTACHMENT_SIZE;
    let left  = textureSampleGrad(height_atlas, atlas_sampler,
        coords + vec2<f32>(-offset,     0.0), atlas_index, ddx, ddy).x;
    let up    = textureSampleGrad(height_atlas, atlas_sampler,
        coords + vec2<f32>(    0.0, -offset), atlas_index, ddx, ddy).x;
    let right = textureSampleGrad(height_atlas, atlas_sampler,
        coords + vec2<f32>( offset,     0.0), atlas_index, ddx, ddy).x;
    let down  = textureSampleGrad(height_atlas, atlas_sampler,
        coords + vec2<f32>(    0.0,  offset), atlas_index, ddx, ddy).x;
    return normalize(vec3<f32>(right - left, f32(2u << atlas_lod) / MAX_HEIGHT,
                               down - up));
}
```

Figure 5.11: This function calculates the surface normal using four samples from the height map.

Chapter 6

Results

Figure 6.1: Depicts Hartenstein (left), a small town, and Dresden (right), the capital of Saxony, rendered using the method presented in this thesis. The renderer is able to reproduce the landscape up close as well as from afar.

This chapter showcases and analyzes the capabilities of the chunked clipmap, the UDLOD algorithm, and the implementation of the terrain renderer. Figure 6.1 depicts two images rendered in real-time, using the method presented in this thesis. As can be seen, the described terrain renderer can scale effortlessly from small villages to vast regions. All performance measurements were taken on an Apple M1 chip equipped with 16 GB of memory.

6.1 Saxony Terrain Data

To test and benchmark this implementation, an extensive terrain data set was required. Consequently, the author chose to utilize the open geodata supplied by the government of the Free State of Saxony [19]. Therefore the entire DTM, DSM, and DOP data was gathered, preprocessed, and partitioned into the format required by the chunked clipmap. Figure 6.2 compares the resulting images when rendering using either the DTM or the DSM as the height data of the terrain.

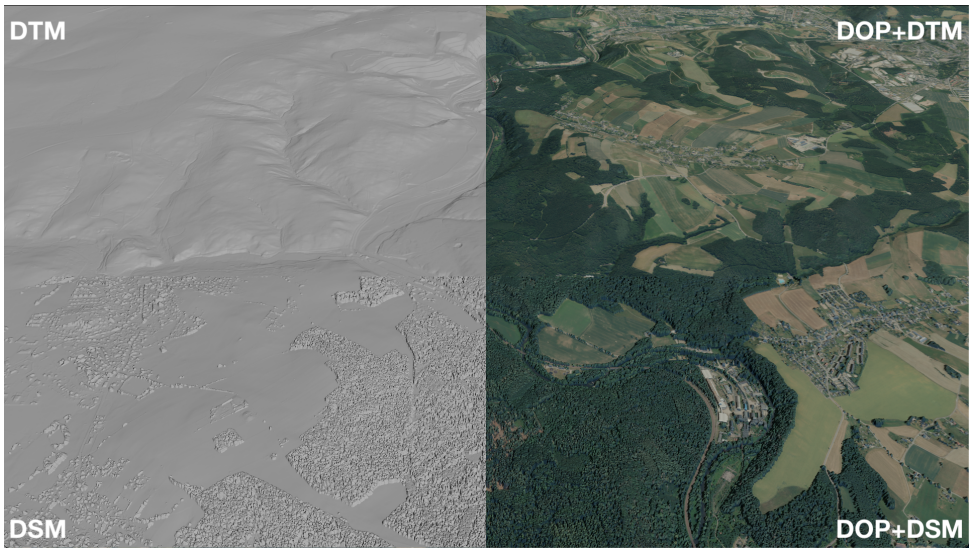


Figure 6.2: A comparison of the rendered images using different types of terrain data. On the left, only the height data is displayed, whereas on the right the terrain is colored according to the DOP information. The top half of the image uses the DTM for rendering, while the bottom one is based on the DSM data.

To demonstrate the capability of the terrain renderer to scale desirably across numerous orders of magnitude, three benchmark datasets were chosen. They encompass the Saxony geodata at a resolution of one pixel per square meter and cover multiple square kilometers. The three datasets are compared in Figure 6.3.

	Hartenstein	Hartenstein and Surroundings	Saxony
Top Down View			
Area	4x4 km (16 km²)	16x16 km (256 km²)	226x170 km (18,416 km²)
Resolution	4,000x4,000	16,000x16,000	226,000x170,000
Size DTM (compressed)	68.7 MB (15.5 MB)	790 MB (261 MB)	70.3 GB (17.3 GB)
Size DSM (compressed)	68.7 MB (23.8 MB)	790 MB (483 MB)	70.3 GB (32.4 GB)
Size DOP (compressed)	103 MB (41.6 MB)	1.19 GB (730 MB)	106 GB (52.8 GB)
Size Total (compressed)	240 MB (81 MB)	2.77 GB (1.47 GB)	246 GB (102 GB)

Figure 6.3: Compares the three datasets that were used to benchmark the terrain renderer.

6.2 Configuring the Chunked Clipmap

Determining a proper configuration of the chunked clipmap parameters is important for achieving high-quality images. Ideally, for each pixel of the frame buffer, one texel of the terrain data should be sampled. Unfortunately, this correspondence is impossible to maintain across the entire screen, due to many factors such as the field of view of the camera, the steepness of the terrain, and the camera's pitch angle. Nevertheless, if enough terrain data is accessible in the chunked clipmap then the rendered image will not look blurry. It turns out that if the resolution of the data stored in each layer of the chunked clipmap is greater than or equal to twice the screen resolution, the terrain will be rendered satisfactory.

In Section 4.5.4, the author explained that sampling compound textures causes issues at the texture borders. This is illustrated by Figure 6.4. The image is zoomed in to better highlight this artifact. To prevent this issue, a texture border was introduced, which stores duplicated information, required by the sampling unit. Testing showed, that a *border_size* of one is sufficient to alleviate this problem for regular sampling. However, due to the normal calculation requiring the data of adjacent texels, a *border_size* of two is necessary to fix these artifacts entirely.

Figure 6.4: Highlights the issue of sampling compound textures. On the left, a *border_size* of zero causes a noticeable seam, between adjacent nodes. On the right, the issue is resolved by using a *border_size* of one.

Depending on the *view_distance*, slight shimmering and aliasing may be observed, even when the anisotropic filtering method, described in Section 4.5.4, is employed. This issue, of oversampling the node textures, is less noticeable, than the texture seams. That is why, the use of additional mip layers, to alleviate this problem, is not always required. However, they can slightly improve the image quality, under certain camera angles.

lod_count	16	border_size	2	720p		1080p		4k			
texture_size	mip_count	center_size	Overhead	max_res	view_distance	max_res	view_distance	max_res	view_distance	node_count	quadtree_size in KB
256	1	252	3.20 %	1280	5.08	1920	7.62	4096	16.25	34	72
256	2	252	29.00 %								
512	1	508	1.58 %	1280	2.52	1920	3.78	4096	8.06	18	20
512	2	508	26.98 %								
1024	1	1020	0.79 %	1280	1.25	1920	1.88	4096	4.02	10	6
1024	2	1020	25.98 %								

Figure 6.5: A table comparing the *view_distance* and quadtree size required to achieve maximum image quality at different screen resolutions. The overhead indicates the percentage of overlapping data required to seamlessly filter the terrain data.

The most important parameter of a chunked clipmap is the *view_distance*. It has to be calculated, depending on the resolution of the frame buffer. Therefore, the maximum resolution, of the x or y direction, has to be divided by the *center_size* of the node. Figure 6.5 shows the ideal *view_distance*, determined for multiple combinations of *texture_size*, *mip_count*, and display resolution. The greater the *mip_count* the less shimmering will be noticeable when looking parallel to the terrain, but this also increases the overhead caused by the redundant storage of node border and mip maps. The *load_distance* parameter has to be specified according to the maximum speed of the viewer to ensure that all data inside the *view_distance* has been loaded.

Figure 6.6 depicts the different LOD layers of the chunked clipmap overlayed on top of the terrain's surface. On the left, the underlying nodes can be seen, and on the right, the blended layers are shown.

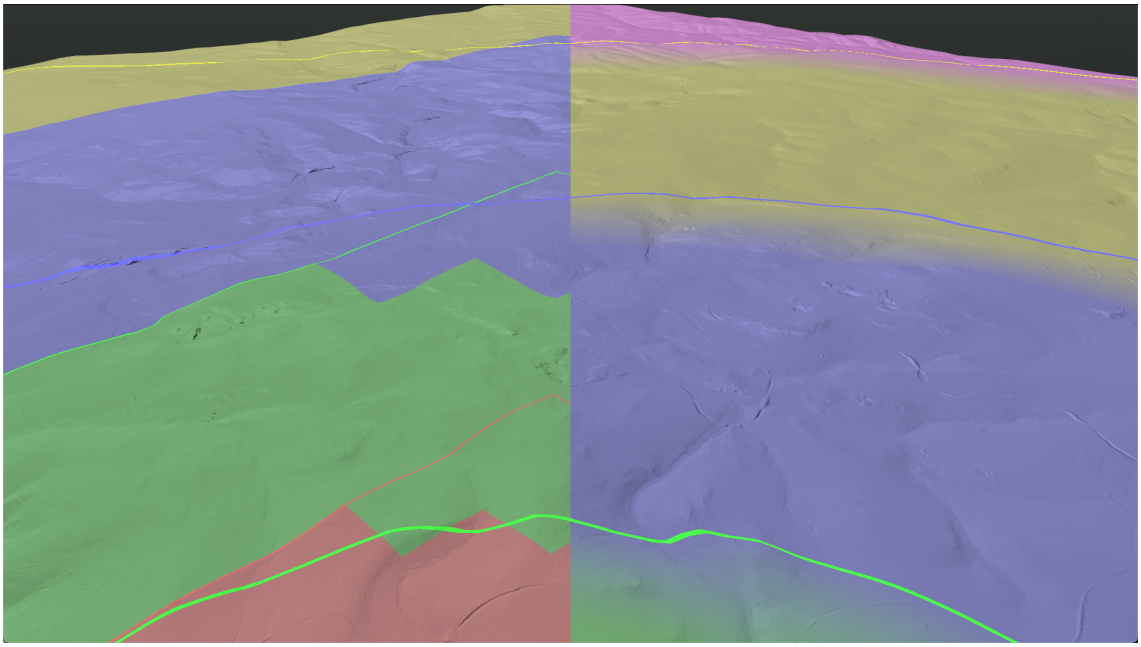


Figure 6.6: Depicts a comparison between the loaded nodes on the left and the blended LOD layers on the right, overlayed on top of the terrain's surface.

6.3 Configuring the Uniform Distance-Dependent Level of Detail

grid_size	tile_scale	pixels_per_quad	prepass_time in µs	vertex_time in µs	fragment_time in µs	gpu_time in ms	vertex_count
2	8	4	450,58	2190,0	8970	11,5	1.453.890
2	16	8	342,0	724,88	5440	6,3	371.028
2	32	16	301,67	232,04	3790	4,34	96.342
2	64	32	298,75	100,25	3020	3,42	26.092
4	16	4	340,75	1650,00	8850	10,59	1.113.016
4	32	8	334,92	516,25	5310	6,13	288.956
4	64	16	312,17	229,46	3780	4,27	78.204
4	128	32	291,54	71,17	3010	3,39	22.060
8	32	4	309,75	1310	8750	10,34	963.102
8	64	8	301,25	447,25	5340	6,03	260.602
8	128	16	300,54	219,88	3780	4,24	73.450
8	256	32	314,08	101,96	3020	3,36	23.008
16	64	4	298,79	1230	8750	10,22	938.074
16	128	8	293,17	431,46	5280	6,00	264.328
16	256	16	285,62	214,88	3740	4,22	82.736
16	512	32	297,50	120,50	3020	3,35	28.952

Figure 6.7: A table comparing the performance of selected combinations of UDLOD parameters.

Configuring the UDLOD algorithm is not as straightforward as it is for the chunked clipmap. Its parameters are largely dependent on the available frame time budget dedicated to rendering the terrain. The quotient of *tile_scale* and *grid_size* determines the approximate side length of each triangle in screen space. This means that for a *pixels_per_quad* ratio of one - every two triangles cover one pixel of the final image. However, rendering that many triangles is unreasonable even on modern hardware, thus a comparison of different configurations is shown in Figure 6.7. A single frame was recorded and measured for each configuration, using the Apple Xcode Profiler. The corresponding mesh for each *pixels_per_quad* value can be examined in Figure 6.8.

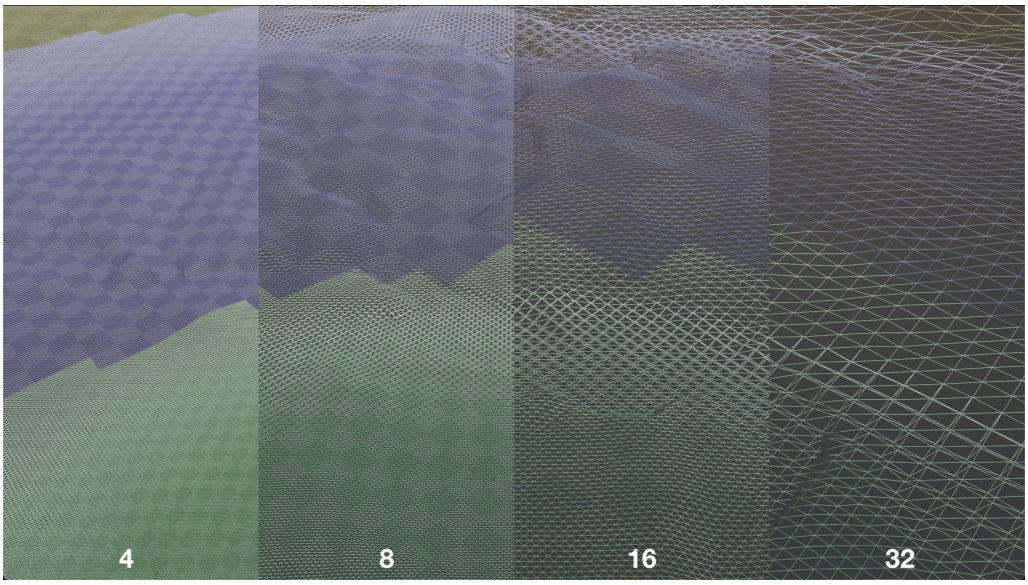


Figure 6.8: Shows the same view rendered using different *pixels_per_quad* ratios side by side.

Analyzing these results of Figure 6.7 uncovers some important considerations for tweaking the

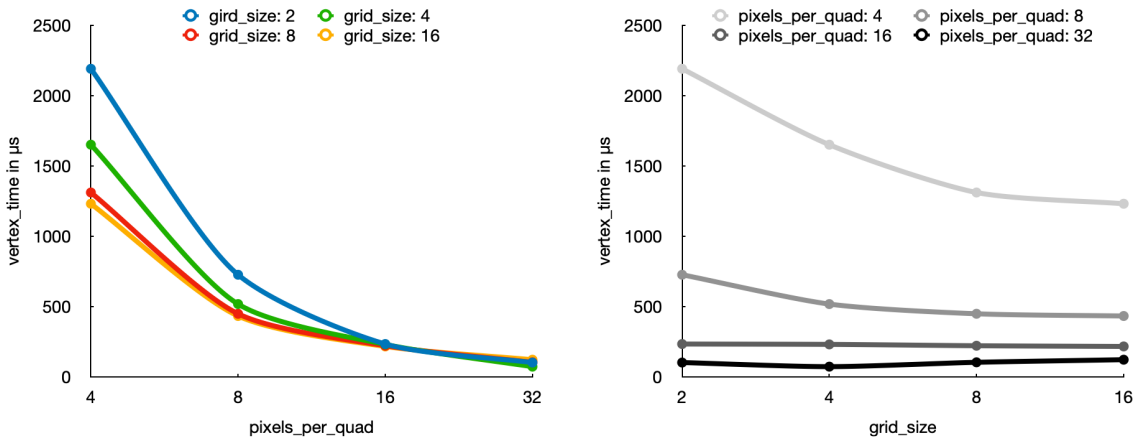


Figure 6.9: Plots the time spent during the vertex shader per frame depending on the *pixels_per_quad* ratio on the left and on the *grid_size* on the right.

UDLOD parameters. Figure 6.9 confirms that the time spent in the vertex shader decreases for increasing *pixels_per_quad* values and thus fewer rendered triangles. Interestingly the same behavior is also true for the fragment shader, as can be seen in Figure 6.10. Because of the relatively small amount of work done in the prepass, the execution time is mostly limited by the compute shader dispatch overhead and not by its actual algorithm. As explained in Section 4.4.3, larger *grid_size* cause less vertex shader overhead and are thus preferable, when utilizing only inexpensive versions of culling. This effect can be observed in the right diagram of Figure 6.9.

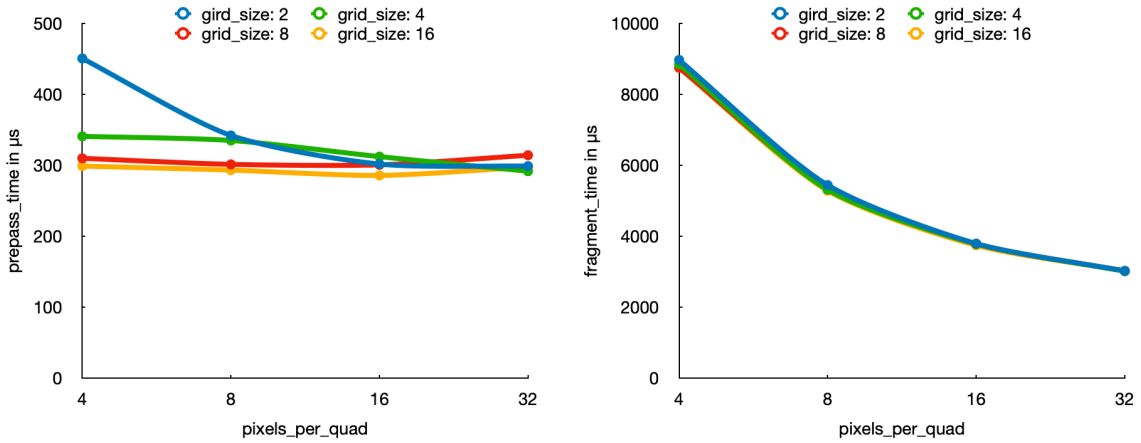


Figure 6.10: Graphs the time spent during the prepass on the left and during the fragment shader on the right depending on the *pixels_per_quad* ratio.

6.4 Evaluation of the Terrain Rending Method

At the beginning of Chapter 4 the author established eight requirements for the "ideal terrain renderer". Now the novel terrain rendering method, consisting of the UDLOD algorithm and the chunked clipmap data structure, will be evaluated against those criteria. To evaluate the final visual quality of the rendered images produced by this implementation, a supplementary video was recorded. It can be watched at `https://youtu.be/ZRMt1GV50nI`. The video showcases additional terrain data from the open geodata of Switzerland [25], to demonstrate the capability of handling steep terrains. Figure 6.11 depicts such an image.

1 Hardware Adaptive Quality Both UDLOD and chunked clipmaps provide multiple parameters allowing for fine-grained tuning of the quality and performance of the terrain renderer. Especially the *view_distance* and the *tile_scale* can be adjusted seamlessly, without having to change anything about the preprocessed terrain data.

2 Unlimited Terrain Size As demonstrated in the prior section, the method is well suited for rendering large-scale terrains with high fidelity in real-time. The limiting factor is the storage capacity required by the terrain data.

3 Seamless Viewing Distances The seamless viewing distances can not be demonstrated using images alone. This chapter showcased the terrain, rendered using many different viewpoints. The auxiliary video was recorded to evaluate the continuous image quality while the camera is in motion.

4 Uniform Screen-Space Error The UDLOD algorithm provides a uniform tesselation in world space, as can be seen in Figure 6.8, which translates well into screen space for most landscapes, and only causes issues for very jagged or steep terrains. However, those kinds of terrains can not be approximated satisfactorily using height maps anyway.

5 Smooth Vertex Morphing The tesselation produced by the UDLOD algorithm is entirely locally and temporally continuous. Every LOD transition is morphed to guarantee the absence of cracks and pops.

6 Random-Access Terrain Data Although this feature is not implemented in the showcased terrain renderer, accessing terrain data on the CPU and the GPU, for purposes other than rendering the terrain, is possible when using the chunked clipmap data structure.

7 Seamless High-Quality Texturing As seen in Chapter 4 and Chapter 5, special considerations were taken to develop a texturing scheme that allows for seamless high-quality trilinear and anisotropic filtering. This is also highlighted in the video.

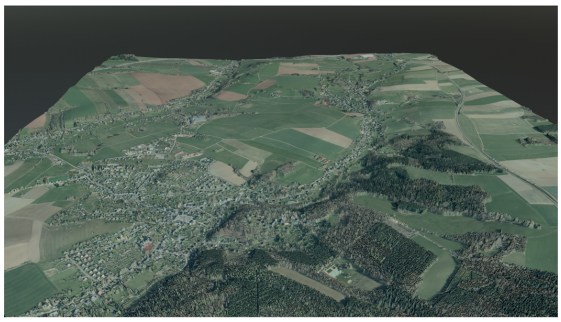
8 Multiple Views Another important quality of the chunked clipmap technique is the ability to accommodate multiple clipped quadtrees, and thus multiple independent views, which can share common terrain data. Although features like split-screen or shadow rendering are not part of the demo application, they are possible, as demonstrated in the video.

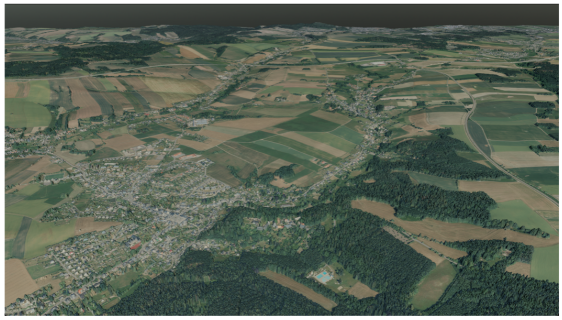
Figure 6.11: Depicts the Swiss Alps, rendered by the terrain renderer.

6.5 Final Benchmark

With the configuration for both the chunked clipmap and the UDLOD algorithm discussed, a final benchmark, between the three datasets, to demonstrate the suitability of the terrain rendering method for large-scale terrains is given. The results can be seen in Figure 6.12. All three terrains can be rendered in real-time at full HD resolution, using 4x multisample anti-aliasing and 16x anisotropic filtering. The parameters were chosen to represent features such as trees and buildings of the DSM accurately, but rendering smoother DEM terrains does not require such expensive parameters.

	vertex_time	fragment_time	gpu_time	vertex_count
Hartentein	2.63 ms	11.8 ms	14.6 ms	2,058,060
Hartentein and Surroundings	6.63 ms	20.5 ms	27.2 ms	5,336,345
Saxony	8.97 ms	24.5 ms	32.9 ms	6,788,698





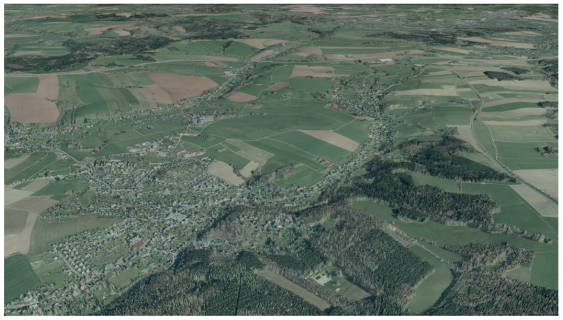


Figure 6.12: Shows a benchmark comparing the results of the same view for all three datasets. The three rendered images correspond to the three terrains and were rendered using a *view_distance* of 4, a *grid_size* of 4, and a *pixels_per_quad* ratio of 2.

Chapter 7

Conclusion and Future Work

This thesis was concerned with two fundamental questions of large-scale terrain rendering, namely "How to represent and manage the terrain data?", as well as, "How to approximate the terrain geometry?". The paper presented an overview of the currently available terrain rendering techniques using the raster graphics pipeline. Based on those previous works, eight requirements of such a terrain renderer were provided and the previous techniques were all found to be unsatisfactory at solving all of them at once. Thus this thesis set out to solve all of them simultaneously.

7.1 Contributions

To solve the two key questions, the author introduced two novel terrain rendering concepts, namely the Uniform Distance-Dependent Level of Detail (UDLOD) and the Chunked Clipmap.

Uniform Distance-Dependent Level of Detail (UDLOD) The Uniform Distance-Dependent Level of Detail (UDLOD) algorithm combines the CDLOD method with the quadtree subdivision of the "Fary Cry 5" terrain renderer. This allows for a seamless and uniform view-dependent triangulation of large planes, which can in turn be displaced into heightfield terrains. Their continuous, fully customizable tesselation density let them adapt perfectly to any quality or performance targets. Additionally, UDLOD provides a great foundation for implementing advanced GPU-based culling techniques to render terrains even more efficiently.

Chunked Clipmap The chunked clipmap constitutes an evolution of the original clipmap data structure, which in turn is an enhancement of the texture mipmap concept. It offers the possibility to represent the data of a virtually unlimited terrain in a view-dependent way. Its chunked representation can be efficiently paged in and out of memory and allows for reusing the same data between multiple views. Additionally seamless trilinear and anisotropic filtering is possible and its implementation was covered by this thesis.

7.2 Future work

Although the terrain rendering method presented in this thesis satisfies all eight requirements it set out to solve, there are still parts that could be improved upon. The programmable hardware-based tessellation stage could be used to tesselate the tiles differently according to a tile-based error metric. Also, the recently introduced mesh shading pipeline could be used to combine both the refinement compute shader of the UDLOD prepass with the rendering of the tiles into the same pipeline. Furthermore, an extension of the chunked clipmap and the UDLOD algorithm to the spherical domain, would allow for rendering high-resolution planets.

Another unsolved question is the real-time modification of the terrain. Due to time limitations while writing this thesis the author could not add this feature to the reference implementation. Although the constraints introduced by the seamless sampling of the terrain data, make updating the terrain data more difficult, it should be possible. Additionally, a more sophisticated node-loading strategy could result in requiring less memory and bandwidth. Especially replacing invisible but loaded nodes, in favor of more impactful ones, should improve the performance and quality of the terrain on memory constraint systems.

Another interesting possibility would be utilizing Technologies like DirectStorage, allowing the node data to be loaded immediately from the hard drive into VRAM without passing it through the CPU.

True-to-life landscape rendering is not limited to just rendering terrain geometry and albedo but includes more challenges, such as shadow rendering, or global illumination, as well as, realistic texturing and shading. The field of terrain rendering poses a wide variety of related, but vastly different problems, such as rendering the sky, vegetation, water bodies, and man-made structures.

Acronyms

RAM	random access memory
VRAM	video random access memory
API	application programming interface
GIS	geographical information systems
FPS	frames per second
TIN	triangulated irregular network
RTIN	right-triangulated irregular network
LEB	longest edge bisection
OOC	out-of-core
LOD	level of detail
DLOD	discrete level of detail
CLOD	continuous level of detail
MLOD	multi-resolutional level of detail
DEM	digital elevation model
DTM	digital terrain model
DSM	digital surface model
DOP	digital orthophoto
ROAM	Real-time Optimally Adapting Mesh
PGM	Persistent Grid Mapping
CDLOD	Continuous Distance-Dependent Level of Detail
CBT	Concurrent Binary Tree
UDLOD	Uniform Distance-Dependent Level of Detail

Appendix

Name	Description
lod_count	number of LOD levels the implicit quadtree possess
tile_scale	continuous scale factor of the tiles used to adjust the quality
grid_size	number of rows and columns of each tile grid
view_distance	determines the distances at which the LOD changes
tile_count	number of tiles to render (length of the final tile list)
vertex_index	index of the current vertex (vertex shader built-in)
tile_index	identifies the tile corresponding to a vertex in the tile list
tile_coords	coordinates of a tile (fetched from the tile list)
tile_size	size of a tile (fetched from the tile list)
grid_index	identifies a vertex inside the grid of the current tile
vertices_per_tile	number of vertices required by each grid/tile
grid_position	the horizontal position of a vertex inside the grid
local_position	the horizontal position of the vertex in local space
morphed_local_position	the morphed position of the vertex in local space

Table 1: An overview of all different parameters and variables required by UDLOD.

Name	Description
lod_count	number of LOD layers of the clipped quadtree
node_count	number of nodes per side of the clipped quadtree
leaf_node_size	size of the smallest nodes (LOD 0) in local space
node_size	size of the nodes adjusted for their LOD in local space
center_size	the size of the area of unique terrain data in the center of each node
border_size	the size of the border of duplicated terrain data around each node
view_distance	determines the distances at which the LOD changes
load_distance	determines the distances at which nodes should be loaded
approximate_height	the height of the terrain under the current viewer position
distance	the approximate distance between the position and the viewer
quadtree_lod	LOD, based on the distance to the viewer, used to access the quadtree
quadtree_coords	coordinates (of the corresponding position) in the quadtree
atlas_index	identifies a corresponding node in the atlas
atlas_lod	identifies the LOD (and thus the size) of the node in the atlas
atlas_coords	coordinates (of the corresponding position) inside the node of the atlas
adjusted_atlas_coords	atlas coordinates adjusted to exclude the border of the node
adjusted_ddx/ddy	the texture gradients adjusted for the node

Table 2: An overview of all different parameters and variables required by the Chunked Clipmap.

```
struct Parameters {
    tile_count: u32,
    counter: i32,
    child_index: atomic<i32>,
    final_index: atomic<i32>,
}
struct IndirectBuffer { workgroup_count: vec3<u32> }
struct Tile { coords: vec2<u32>, size: u32 }
struct TileList { data: array<Tile> }
struct NodeLookup { atlas_lod: u32, atlas_index: i32, atlas_coords: vec2<f32> }
struct Blend { lod: u32, ratio: f32 }

// uniform values:
var<uniform> LOD_COUNT:          u32;
var<uniform> NODE_COUNT:         u32;
var<uniform> VIEW_POSITION:      vec4<f32>;
var<uniform> FRUSTUM_PLANES:     array<vec4<f32>, 6>;
var<uniform> MAX_TILE_COUNT:     u32;
var<uniform> TILE_SCALE:         f32;
var<uniform> VERTICES_PER_TILE:  u32;
var<uniform> VERTICES_PER_ROW:   u32;
var<uniform> GRID_SIZE:          f32;
var<uniform> APPROXIMATE_HEIGHT: f32;
var<uniform> MAX_HEIGHT:         f32;
var<uniform> LEAF_NODE_SIZE:     f32;
var<uniform> BLEND_DISTANCE:     f32; // view_distance * leaf_node_size
var<uniform> BLEND_RANGE:        f32;
var<uniform> MORPH_DISTANCE:     f32; // view_distance * leaf_node_size
var<uniform> MORPH_RANGE:        f32;

// for each attachment:
var<uniform> ATTACHMENT_SIZE:    f32; // center_size + 2 * border_size
var<uniform> ATTACHMENT_SCALE:   f32; // center_size / ATTACHMENT_SIZE
var<uniform> ATTACHMENT_OFFSET:  f32; // border_size / ATTACHMENT_SIZE
var attachment_atlas: texture_2d_array<f32>;

// prepass bindings
var<storage, read_write> indirect_buffer: IndirectBuffer;
var<storage, read_write> parameters: Parameters;
var<storage, read_write> temporary_tiles: TileList;
var<storage, read_write> final_tiles: TileList;
var quadtree: texture_2d_array<u32>;

// render bindings
var<storage> tiles: TileList;
var quadtree: texture_2d_array<u32>;
var atlas_sampler: sampler;

fn node_size(lod: u32) -> f32 { return f32(LEAF_NODE_SIZE * (1u << lod)); }

fn approximate_world_position(local_position: vec2<f32>) -> vec4<f32> {
    return vec4<f32>(local_position.x, APPROXIMATE_HEIGHT, local_position.y, 1.0);
}
```

Figure 1: Depicts the structures, uniform parameters, bindings, and helper functions used in the code snippets of Chapter 5.

Additional Material

Accompanying this thesis, the author provides a reference implementation of the introduced method, as described in Chapter 5. The source code can be found at the following link: https://github.com/kurtkuehnert/terrain_renderer. This repository contains the instructions for compiling two executables. One of which downloads the desired terrain dataset from the geodata portal of Saxony [19] or that of Switzerland [25]. The other is the demo terrain renderer. It can visualize the datasets downloaded by the former tool. For further information, please refer to the instructions provided in the repository.

To better showcase the capabilities of the terrain renderer, a supplementary video was recorded. It can be found at https://youtu.be/ZRMt1GV50nI.

References

[1] Raphaël Lerbour. "Adaptive streaming and rendering of large terrains". PhD thesis. Université Rennes 1, 2009.

[2] Michael Garland and Paul S Heckbert. *Fast polygonal approximation of terrains and height fields*. 1995.

[3] Jonathan Dupuy. "Concurrent Binary Trees (with application to longest edge bisection)". In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.2 (2020), pp. 1–20.

[4] Mark Duchaineau et al. "ROAMing terrain: Real-time optimally adapting meshes". In: *Proceedings. Visualization'97 (Cat. No. 97CB36155)*. IEEE. 1997, pp. 81–88.

[5] David Luebke et al. *Level of Detail for 3D Graphics*. 2003.

[6] Thatcher Ulrich. *Rendering Massive Terrains using Chunked Level of Detail Control*. 2002.

[7] Peter Lindstrom et al. "Real-time, continuous level of detail rendering of height fields". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, pp. 109–118.

[8] Martin Hedberg. *Rendering of Large Scale Continuous Terrain Using Mesh Shading Pipeline*. 2022.

[9] Egor Yusov. *GPU Pro3 - Real-Time Deformable Terrain Rendering with DirectX 11*. 2018.

[10] Willem H De Boer. "Fast terrain rendering using geometrical mipmapping". In: *Unpublished paper, available at http://flipcode.com/articles/article_geomipmaps.pdf* (2000).

[11] Filip Strugar. "Continuous distance-dependent level of detail for rendering heightmaps". In: *Journal of graphics, GPU, and game tools* 14.4 (2009), pp. 57–74.

[12] Jeremy Moore. *Terrain Rendering in 'Far Cry 5'*. 2018.

[13] Daniel Wagner. "Terrain geomorphing in the vertex shader". In: *ShaderX2: shader programming tips and tricks with DirectX 9* (2004), pp. 18–32.

[14] Tomas Akenine-Möller et al. *Real-Time Rendering Fourth Edition*. 2018.

[15] Ulrich Haar and Sebastian Aaltonen. *GPU-Driven Rendering Pipelines*. 2015.

[16] Kai Ninomiya, Brandon Jones, and Myles C. Maxfield. *WebGPU W3C Working Draft*. 2022. URL: https://www.w3.org/TR/webgpu/.

[17] Christopher C Tanner, Christopher J Migdal, and Michael T Jones. "The clipmap: a virtual mipmap". In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 1998, pp. 151–158.

[18] Louise Croneborg et al. *DIGITAL ELEVATION MODELS*. 2015.

[19] Staatsbetrieb Geobasisinformation und Vermessung Sachsen [GeoSN]. *Offene Geodaten Sachsen (DOP20, DGM1, DOM1)*. 2022. URL: https://www.geodaten.sachsen.de/.

[20] Frank Losasso and Hugues Hoppe. "Geometry clipmaps: terrain rendering using nested regular grids". In: *ACM Siggraph 2004 Papers*. 2004, pp. 769–776.

[21] Arul Asirvatham and Hugues Hoppe. "Terrain rendering using GPU-based geometry clipmaps". In: *GPU Gems 2* (2005), pp. 27–45.

[22] Yotam Livny et al. *Persistent Grid Mapping: A GPU-Based Framework for Interactive Terrain Rendering*. Jan. 2007.

[23] Kevin Brothaler. *Android Lesson Eight: An Introduction to Index Buffer Objects (IBOs)*. 2012. URL: http://www.learnopengles.com/android-lesson-eight-an-introduction-to-index-buffer-objects-ibos/.

[24] *Geometric Approach – Testing Boxes II*. 2011. URL: http://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-testing-boxes-ii/.

[25] Federal Office of Topography [swisstopo]. *Open Geodata Switzerland (DTM2, DOM2)*. 2022. URL: https://www.swisstopo.admin.ch/.