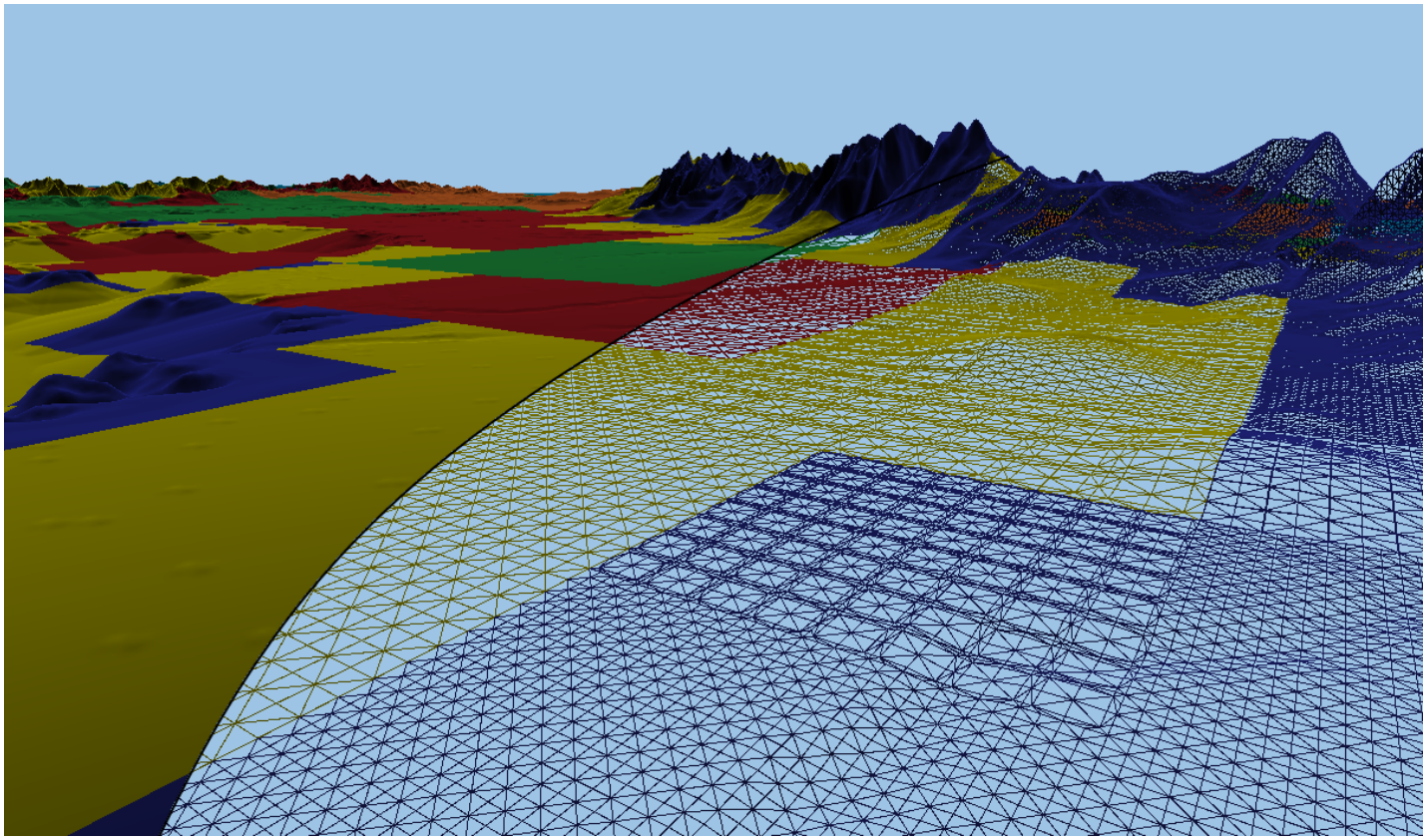


Martin Hedberg

Rendering of Large Scale Continuous Terrain Using Mesh Shading Pipeline

External supervisor	Emil Lindberg
Supervisor	Alexandre Bartel
Examiner	Henrik Björklund

Spring 2022
Master Thesis, 30 ECTS
M.Sc.Eng in Computing Science, 300 ECTS

UMEÅ UNIVERSITET

Abstract

Large open terrains require visual simplifications to be able to be rendered in real-time. One possibility is to subdivide the terrain into a quadtree where each node in the tree represents a simpler version of a tile of the terrain. This requires an algorithm that can traverse the quad tree, choose which tiles to render, and to save performance, remove tiles that are not in front of the camera. When the terrain is simplified in this manner, the algorithm must also make sure that the terrain does not feature holes in between the simplified parts. This thesis investigates the differences between the Mesh shading pipeline and the traditional graphics pipeline in two kinds of these terrain algorithms. First a naive algorithm implemented on the GPU is measured in terms of execution time, it is then changed to use a Concurrent Binary Tree for the traversing of the quad tree and measured to see if this data structure could be utilized to improve the algorithm in terms of execution time. The results show that, while the Concurrent Binary Tree yields a theoretical improvement to a Quad Tree based terrain simplification, the practical application is limited. Regarding the choice of rendering pipeline, the Mesh Shading pipeline did not yield any significant improvements and a conclusion could be drawn that, without any advanced culling algorithms, the traditional pipeline performed better overall. This thesis also presents a continuous rendering algorithm, which draws inspiration from two state of the art rendering algorithms.

Keywords: Terrain, Continuous, Rendering, Quad Tree, Mesh Shading, Concurrent Binary Tree

Acknowledgement

Carl XVI Gustaf - *The King of Sweden*

Ulla Hedberg - *Grandmother and supporter*

Adrián Erik Clarenbeek Sánchez - *For being my Palpatine*

Filippa Dahlgren Lidman - *Coffee and rants*

Emil Söderlind - *For being a great mentor and father figure in my life*

Emil Lindberg - *Supervisor and over the shoulder observer*

Alexandre Bartel - *University supervisor*

Olle Sundin - *My Lord*

Contents

List of Figures

List of Tables

1 Introduction

> Let there be stunning terrain.
>
> *God*

Today, it is hard to imagine a world without virtual environment. Games, movies, and other media is dependent on the use of these environments. These environments can look and act in many ways, but one of the most stunning and at the same time hardest to render might be virtual terrains.

A terrain might cover thousands, or even millions, of square kilometres, but requires a precision of centimetres when viewed up close. That is why careful considerations must be made from the developers of these terrain systems.

While we might take virtual terrain for granted when playing a video game or using Google Earth, a lot of work is needed for them to look good while still having a frame rate high enough to not break the user's immersion. To achieve this balance of visual quality and frame rate, simplifications of the terrain visualization must be made.

Visual simplifications of terrains spawned a lot of research in the early days of computer graphics, and while most algorithms were developed in the early 90s, they are still very relevant. The basic idea behind most algorithms is the same. Areas further from the user will not need to be as rich in detail as the areas further away. However, to not break the immersions, these algorithms must make sure that the simplifications are not noticeable by the user but stay as aggressive as possible in removing unnecessary rendering.

The responsibility of keeping the framerate high enough is therefore not only put on the graphics card that renders the terrain but also on the algorithms that compute the simplifications. Therefore, this thesis will examine current state of the art terrain simplification and rendering methods, then implement and evaluate their performance against each other.

This thesis is made in collaboration with Paradox. Since most of Paradox's games revolves around strategy on large maps, it is crucial for them to have terrain that perform well so they can continue to enchant and entertain millions of people.

1.1 Motivation

In the last couple of years, published research regarding terrain simplifications has been few, most of them date back more than a decade [21, 5, 4, 16, 13, 23]. These older papers do not usually utilize the GPU to its fullest extend for culling and Level of Detail selection, since this is a rather new feature in regards to when the papers were released. Also, there are a lot of game studios working with this, but usually they keep their findings private as a trade secret instead of publishing their research.

The main motivation of this thesis is therefore to investigate the current state of terrain simplification algorithms for GPU usage.

1.2 Objectives

The main objective of this thesis is to implement two Level of Detail algorithms and compare them on both the traditional graphics pipeline and the new Mesh shading pipeline. The algorithms are to be fully implemented on the GPU.

Aim 1. Evaluate the use of Concurrent Binary Trees in Terrain rendering

No.	Objectives
1	Investigate current generation Level of Detail algorithms.
2	Choose algorithm that can use Concurrent Binary Tree.
3	Implement and evaluate.

Aim 2. Evaluate the Mesh shading pipeline in Terrain rendering.

No.	Objectives
1	Investigate current generation rendering methods.
2	Implement and evaluate.

Aim 3. Present a continuous render algorithm.

No.	Objectives
1	Identify criteria for continuous rendering.
2	Evaluate algorithms from literature.
3	Implement example.

1.3 Paradox Interactive

Paradox Interactive is a video game developer and publisher founded in 2004. Paradox are mostly focused on making games called Grand Strategy Games, but have some games in other genres. A few of Paradox most well-known titles include Cities: Skyline, Stellaris, Crusader Kings, and Hearts of Iron.

This thesis was made at Paradox Arctic, which is their northernmost studio in Sweden, located in Umeå.

2 Theory

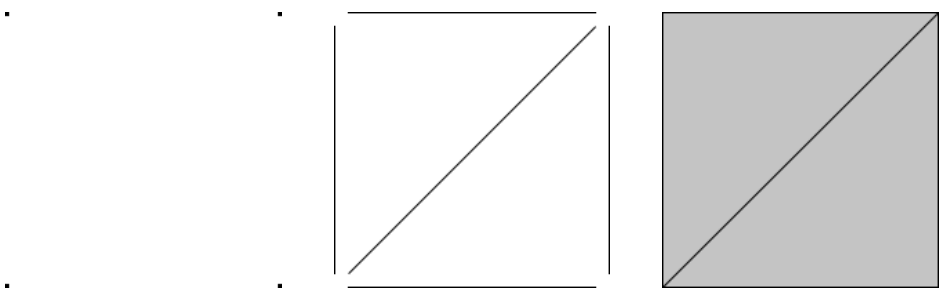
This chapter presents the underlying theory that the thesis is built upon. Section 2.1 explains some of the fundamentals of computer graphics and Section 2.2 covers the traditional graphics pipeline. Section 2.3 covers the newer Mesh Shading pipeline and Section 2.4 covers Concurrent Binary Trees.

2.1 Computer graphics basics

This section explains some of the fundamentals in the field of computer graphics.

2.1.1 Polygonal mesh

In 3D environments such as video games, 3D objects are represented by what is called a **mesh**. A mesh is a collection of vertices, edges and faces that together defines a shape of the 3D object [18, 14, 1].



(a) Vertices (b) Edges (c) Faces

Figure 1: A example of a square mesh's components

Figure 1 shows a mesh of a square. It consists of 4 vertices (a) and 5 edges (b). Together, these form 2 triangles, or 2 faces (c) as they are called in computer graphics. Triangles have many advantages over other shapes, but most noticeably is that they will always be a flat surface, while a square has the possibility to be concave or convex. Triangles may sometimes be referred to as primitives, since they are the most basic element. However, it is worth noting that other shapes can be primitives, for example a point or a line.

2.1.2 View frustum

The **view frustum** is the region that encapsulates what the camera sees. It is a box with 6 sides, two horizontal sides (left and right), two vertical sides (top and bottom), one near side and one far side (opposite to near side). An example of a view frustum can be seen in Figure 2.

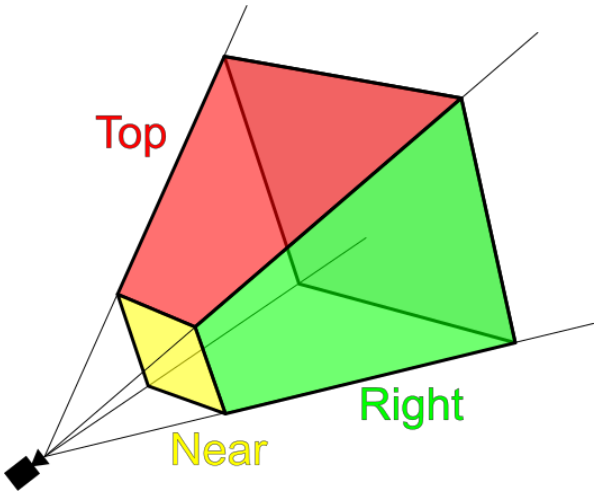


Figure 2: A view frustum

2.1.3 Culling

In computer graphics, culling is the term used to describe the procedure of removing unnecessary parts before rendering. Some common culling operations include back-face culling, where the back of a face is not rendered; view frustum culling, where object outside of the view frustum are discarded before any rendering operations begin; and occlusion culling, where objects occluded by other objects are discarded before any rendering operations begin.

2.1.4 Mipmaps

Mipmaps are images that have been pre-computed to have gradually lesser resolution versions of itself. Mipmaps consist of a finite amount of levels, where level 0, or mip 0, is the lowest resolution of the original image.

Figure 3: A mipmap image with 5 levels

Figure 3 represent a mipmap image with 5 levels of mipmap. Level 4, which is the original image, is 512x512 pixels in size. The image resolution gets halved every mip level, such that mip 0 is 32x32 pixels in size.

2.2 The graphics pipeline

To be able to render meshes to the screen, a standardized way to send and compute data on the graphics card is needed. This is where a graphic pipeline comes in. A graphics pipeline is a set of stages that describe how data flows through the graphics card. There are different graphics pipeline specifications depending on the graphics API, but they all more or less represent the same things, albeit with different names for certain stages. Three of the biggest graphics APIs are Vulkan[1], OpenGL[2], and DirectX[3]. From here on out, when referring to a pipeline the Vulkan naming standard will used.

2.2.1 The traditional graphics pipeline

During the past 20 years, what we will refer to as the traditional pipeline has been in use. It is widespread and exists on all modern graphics cards. The traditional pipeline is broken down into 7 steps, with some being fixed function, some being programmable **shader** programs, and some being a mix of both. Fixed functions are functions decided at hardware level while shader programs are programs that are executed on the GPU.

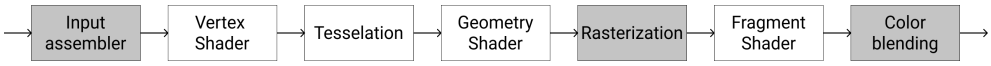


Figure 4: The traditional rendering pipeline, white boxes are programmable stages while gray boxes symbolises a fixed function stage

In Figure 4, an overview of the traditional pipeline can be seen. A short summary of each stage of this pipeline follows:

1. Input assembler - Receives data and assembles it into the correct primitive (triangles, lines, etc.). Fully fixed function.
2. Vertex shader - Programmable shader stage that operates on each vertex. Can for example displace vertices based on some criteria.
3. Tessellation - Subdivides the mesh into more vertices and faces. Consists of three substages. This stage is optional.
   (a) Control Shader - Programmable stage used to control how much tessellation a certain patch will get. Hull shader in DirectX.
   (b) Primitive Generator - Fixed function that does the subdivision. Often referred to as hardware tessellation.
   (c) Evaluation Shader - Last stage of tessellation and programmable. Uses the abstract patch data generated in the previous stage and computes it into a vertex. Domain shader in DirectX.
4. Geometry shader - An optional programmable stage. Takes a primitive as input and can output zero or more primitives.
5. Rasterization - Fixed function that breaks up a primitive into pixel fragments.

[1] https://www.khronos.org/registry/vulkan/specs/1.3/pdf/vkspec.pdf
[2] https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf
[3] https://docs.microsoft.com/en-us/windows/win32/direct3d12/pipelines-and-shaders-with-directx-12

6. Fragment shader - Programmable shader that operates on each fragment.
7. Color blending - Fixed function that blends fragments together into a final pixel.

However, this pipeline has some limitations. When rendering a very high-definition mesh, there might be a lot of unnecessary work done since only a limited amount culling can occur. Also, while the pipeline is multi-threaded, the division of work might be limited and bottlenecks can occur in fixed function stages [10].

2.2.2 Compute shader

A **compute shader** is a shader used to do parallel computing on the GPU. It gives the developer a lot more control over the GPU and its resources, e.g., by specifying how many threads to be used. The data computed on the compute shader can either be consistent on the GPU, i.e., stored and used in later in vertex shading or another compute shader dispatch, or sent back to the CPU. One drawback with the compute shader is that it cannot dispatch any triangles to the framebuffer for rendering. Because of this, a new shader pipeline has been introduced called the Mesh Shading pipeline [10].

2.2.3 Instanced rendering

When a mesh is ready to be rendered to the screen, a draw call must be issued. The draw call tells the GPU that it can execute and render the data it has received. For performance reasons, draw calls should be kept at a minimum when drawing a lot of meshes. When the same mesh needs to be rendered, the GPU can use a single draw call to render the mesh multiple times. This is what is referred to as **instance rendering**, or sometimes batched rendering. This method makes it so that the same mesh can be rendered multiple times quite efficiently [1, p.797].

2.3 Mesh Shading pipeline

To address limitations in the traditional linear pipeline, Nvidia presented their new task and mesh shader in 2019 [17]. This pipeline aims to give more control over how the GPU operates to the graphics programmer. It resembles the Compute shader pipeline in the sense that the developer is in greater control over workload division, but the mesh shader takes things one step further and allows primitives to be generated and sent to the rasterizer directly on the GPU.

The new mesh shading pipeline consists of fewer steps than the traditional pipeline. The two major steps are the task shader (amplification shader in DirectX 12) and the mesh shader [8].

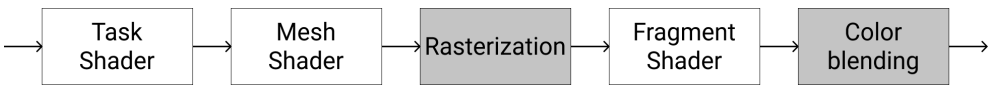


Figure 5: Overview of the Mesh shading pipeline

In Figure 5 an overview of the mesh shading pipeline can be seen. White stages are programmable shaders while gray stages are fixed function. It is worth noting that

only the task and mesh shading stages are unique for the mesh shading pipeline, the rasterization stage and its predecessors are the same as in the traditional pipeline.

2.3.1 Mesh shader

A mesh shader consists of one or many workgroups, where each workgroup consists of threads that work together to generate primitives. Each workgroup has a unique group id assigned to it, and each thread has an id unique to that workgroup. Input to the mesh shader is managed by having a payload for each workgroup where the data is shared by the workgroup's threads. This payload can consist of arbitrary data the developer chooses [10]. Some technical limitations for each mesh shader workgroup:

Payload size	16kB
Workgroup threads	128
Output vertices	256
Output primitives	256

Table 1: Mesh shader workgroup limitations

Because most real meshes consist of more than 256 primitives, the mesh must be broken down into smaller pieces called **meshlets**.

2.3.2 Task shader

The task shader is an optional expansion of the mesh shading pipeline. It is intricately linked to a compute shader, as it is remarkably similar but with the extra addition that it can dispatch mesh shader workgroups. Like the compute and mesh shader, the task shader consists of workgroups, where each workgroup is a collection of threads. Each workgroup has a shared payload of arbitrary data that all threads can utilize [10]. Some technical limitations for task shader workgroups:

Payload size	16kB
Workgroup threads	128
Mesh shaders dispatches	64k

Table 2: Task shader workgroup limitations

Task shaders can be used to do the work that control and evaluation shader previously was responsible for. However, more commonly the task shader is used to do work that compute shaders used to do such as culling of the geometry.

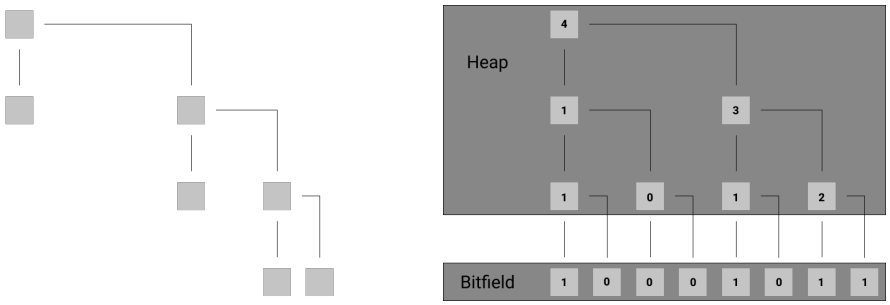
2.3.3 Per-primitive attributes

Another feature that the Mesh Shading pipeline introduces, is the ability to have per-primitive attributes [9]. Here it is possible to define custom attributes, or use pre-defined attributes. One attribute of particular interest is the ability to mark the primitive for culling. This triggers the GPU to discard the primitive before it is sent to rasterization, and can therefore save performance if the fragment shader is performance intense.

2.4 Concurrent Binary Tree

A Concurrent Binary Tree (**CBT**) is a novel data structure presented by Dupuy in 2020 [6]. A CBT is able to represent the shape of any full binary tree such that each leaf-node in said tree can be efficiently operated upon concurrently. The depth of the CBT will never change after initiation, therefore the CBT can be represented as an array implementation of a binary heap in memory instead of using pointer. This is the reason why a CBT is well suited for GPU usage.

A CBT is compounded by two components, the heap and a bitfield. The bitfield is a way to efficiently encode and store the data of the CBT, whereas the heap is used to organize and keep track of all leaf nodes. A visual illustration of this can be seen in Figure 6.



(a) Binary tree (b) Encoded CBT

Figure 6: A binary tree of depth 4 and the CBT encoded from it.

The bitfield is an array of binary numbers. An entry of 1 means a leaf node, where the amount of 0 following the 1 represents the height of the leaf node. The heap is stacked on top of this bitfield, i.e., the bottom layer of the heap is the bitfield itself. The heap is then the sum of all underlying nodes, and therefore the root node of the heap holds the current amount of leaf nodes in the CBT.

Since the tree is represented by the bitfield, splitting and merging of nodes is done by flipping bits in the bitfield. These operations can thus be made thread safe by simply using atomic read and write operations and their time complexity is $O(1)$.

2.4.1 Thread dispatching

Another strength of the CBT is the possibility to dispatch one thread for each leaf node. Since the heap stores the amount of leaf nodes in the root node, the exact amount of threads needed can be dispatched. When dispatched, the thread can decode the leaf node belonging to that thread by doing a form of binary search on the heap that depends on the thread ID. Consequently, thread dispatching has the time complexity of $O(d)$, where d is the depth of the CBT.

2.4.2 Sum-reduction

To update the CBT's heap after split and merges has occurred, Dupuy presents a sum-reduction algorithm which adds the two children and stores them in the parent,

working bottom-up starting at level depth-1. This algorithm is then made concurrent by executing level by level in serial, while each level is divided on a per node basis between threads.

This algorithm is later improved upon by Yao at Unity Technologies [20]. Because the bottom level is represented as bits, the initial algorithm will suffer from data lock conflicts. To mitigate this, Yao chose to let threads work on 32-bit chunks of the bitfield instead, counting all the bits and applying it to the node 5 levels above. The performance gained from not building the 5 bottom levels heavily outweighs the performance lost from having to manually search through 32 bits in the bitfield.

2.4.3 Update routine

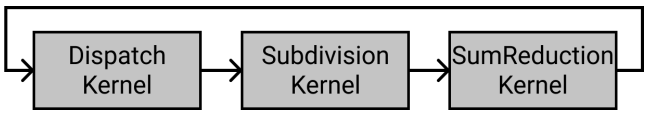


Figure 7: CBT update routine

A typical update routine for a CBT is displayed in Figure 7. First, the dispatch kernel fetches how many threads should be launched by reading the value of the root node in the CBT. From that, a dispatch can be made to launch the subdivision kernel with the right number threads. In the subdivision kernel, nodes can be merged and/or divided concurrently. Lastly, the sum-reduction kernel, which executes the sum-reduction routine described in section 2.4.2, is dispatched.

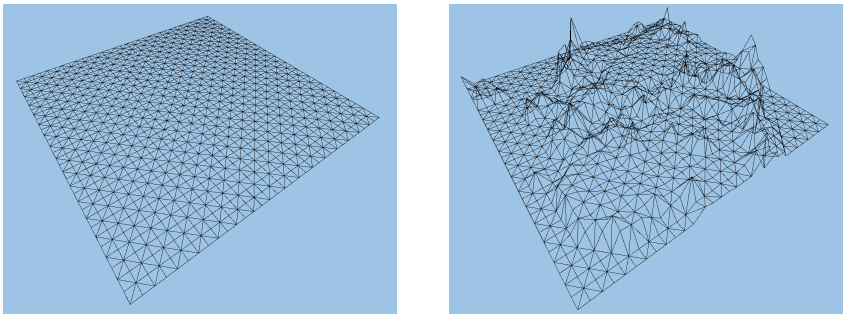
3 Literature Survey

The following literature survey serves as a starting point in the topic that is terrain rendering. The goal of the survey is to investigate the topic and find a feasible method that can be fully implemented on the GPU, be accelerated with a Concurrent Binary Tree and be used with the Mesh shading pipeline as outlined by the aims of the paper.

3.1 Terrain in video games

A quite common way of representing a terrain is by using a **heightmap**. A heightmap is an image that stores discrete height values at each pixel. By using this image, a flat surface can map the pixel values from the heightmap to points on the surface and displace those point in the y-axis by the value from the heightmap.

3.1.1 Regular grids



(a) Flat (b) Displaced

Figure 8: A regular grid, left is a flat surface while right has been displaced

A naive approach for rendering a terrain from a heightmap would be to simply convert each pixel in the heightmap to a uniform grid of vertices in world space. Faces can then be constructed by connecting these vertices in some uniform way to form a flat surface that can be displaced by the heightmap. This would be referred to as a **regular grid**, see Figure 8, where (a) is a flat regular grid and (b) is a regular grid displaced by a heightmap. Although this approach would work on smaller terrain because of the immense computing power we have today, this will not scale with larger terrains, even with today's computing power. The number of vertices would scale exponentially with the area of the terrain [14].

3.1.2 Triangular irregular networks

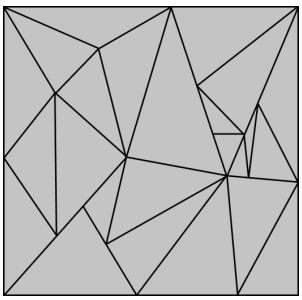


Figure 9: A triangulated irregular network

One way to increase the performance with only a marginal loss of visual accuracy is by irregularly distribute the vertices and faces for a minimal use of resources, e.g., a flat field would need less vertices to be accurately visualized then an irregular mountain. This would create what is referred to as a triangulated irregular network (**TIN**). One drawback of TINS is the amount of precomputing required compared to regular grids [7].

3.2 Level of Detail

Level of Detail (**LOD**) is the term used to describe the amount of simplification a given mesh has been exposed to. A high Level of Detail means that the mesh has been exposed too little to no simplification while a low Level of Detail means that the mesh has been simplified. The reason that LOD exists is to simplify meshes that has a lower importance in order for frames to be rendered faster [1].

To decide which level of detail the mesh should be rendered in, the developer can choose whatever measurement they need. One common metric is to use the distance from the mesh to the camera and measure the perceived error caused by the simplification, referred to as **screen-space error**.

3.2.1 Discrete LOD

Discrete LOD is the traditional type of LOD. It has discrete steps of simplification and is well suited for modern hardware. It is usually used for static meshes where the LOD simplifications are pre-calculated and only the decision of which LOD to choose is done at runtime. Because of the discrete nature, the LOD transition is sudden and can be quite noticeable for the user [14, 1].

One way to mitigate the sudden switching in discrete LOD is to blend between two LOD levels when switching. This results in a smoother transition but might impact the performance since two versions of the model needs to be present, which defeats the purpose of LOD [1].

3.2.2 Continuous LOD

Continuous LOD (**CLOD**) differs from the traditional discrete LOD in that CLOD is not defined as discrete steps. CLOD is built by having a spectrum of detail between levels encoded into a data structure and extract the LOD at run-time. This leads to a much smoother transitioning between different LOD levels [14].

3.3 Multi-resolution terrain

A **multi-resolution** mesh can be defined as a spatial data object that consist of a base mesh. This base mesh will be of the coarsest approximation, i.e., the highest LOD that the object accepts. A set of updates can then be applied to the base mesh to spatially update the LOD of the mesh [4].

These meshes are well suited for terrain rendering since terrain can span vast areas, parts of less significance can be of lower LOD and therefore consume less resources. Multi-resolution meshes can also query parts of the mesh data during run-time, which is useful since map data can consume hundreds of gigabytes of memory [16].

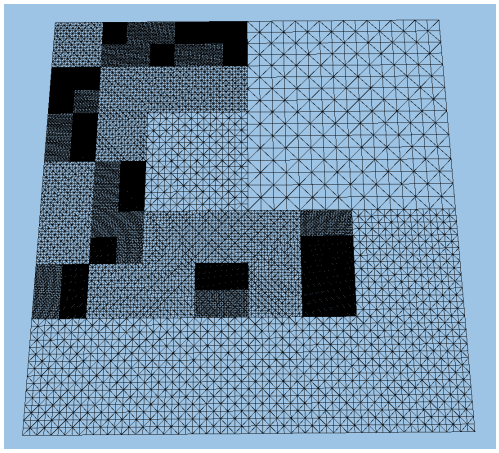


Figure 10: Multi-resolution terrain mesh

Figure 10 is an example of a multi-resolution terrain where various parts are of different resolutions. Each area is in turn a regular grid of different scales.

3.3.1 Cracks and T-junctions

When two areas of different resolutions meet, a gap in the terrain can occur. This is referred to as cracks. There are varied reasons why cracks can occur. In a multi-resolution terrain of regular grids, they occur when a vertex of a lower LOD area does not align with the edge of a higher LOD area [3].

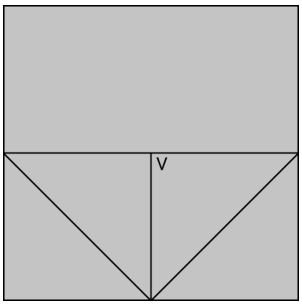


Figure 11: A T-junction is present at the vertex V

If the vertex of a lower LOD area line up with the higher LOD areas edge, something referred to as a T-junction occurs. They are the product of when two edges form a T-shape, see Figure 11 for a visual representation. Because of floating point rounding errors, T-junctions can also create gaps in the terrain. By removing T-junctions in multi-resolution terrain consisting of regular grids, cracks and gaps would be eliminated and therefore result in a continuous terrain [3].

In a multi-resolution terrain of TINS, cracks would occur between TINS if the edge vertices of two TINS do not align with each other. This can be a major challenge for TINS in a multi resolution network since the neighbors are not known until runtime [2].

3.3.2 Out-of-core rendering

Out-of-core rendering (**OOC**) is the term used when not all render data is loaded into memory at once but is fetched from a secondary storage when needed. There are several reasons why one would want to utilize OOC when working with terrain rendering. The main reason is that it is sometimes not possible to load all terrain data into working memory at once since it can reach giga- or even terabytes in size. Another reason is that it could increase loading times of the application if large datasets must be loaded every time the application is launched [3].

3.4 Terrain LOD algorithms

There exists numerous LOD algorithms, each with their own advantages and disadvantages. Most of these algorithms are usually an adaptation of either Geometry Clipmaps or Chunked LOD.

3.4.1 Geometry Clipmap

Geometry Clipmap is a LOD algorithm which is straightforward to implement. It functions by representing the terrain as multiple regular grids stacked on top of each other centered around the camera. The grids' size is getting increased by the power of 2 for each level, to form a pyramid of grids around the camera, see Figure 12. The grids follow the camera as it moves and can be efficiently updated in the vertex-shader [13].

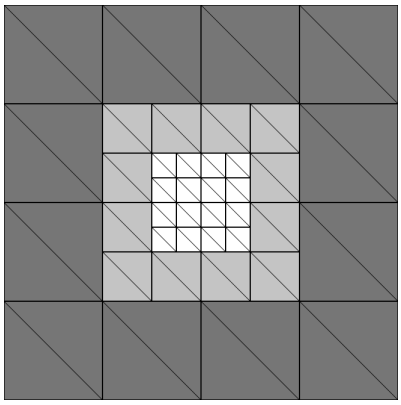


Figure 12: A geometry clipmap plane of 3 levels

Geometry Clipmaps is a great, GPU friendly rendering method, with advantages such as seamless blending between levels, no pre-processing of the terrain and it is easy to implement. However, it has some drawbacks. Most noticeable being the higher triangle count to represent certain terrain compared to other algorithms and the loose screen-space error guarantees. The higher triangle count is contributed to by the fact that Geometry Clipmap assumes the worst-case terrain, i.e., that the terrain is very uneven throughout the entire terrain. This is rarely the case however, since typical outdoor areas usually have a mix of both even and uneven terrain [3].

Since Geometry Clipmap do not calculate any screen-space error, there cannot be any guarantees regarding the bounds of the screen-space error. This means that the screen-space error can be especially visible in certain scenarios, e.g., areas close to the camera might be flat and therefore have a low screen-space error and at the same time areas further away from the camera might be mountainous and have a large screen-space error. There are algorithms that try to use other measurements to get a more accurate measurement of the screen-space error, for example using Summed Area tables [12].

3.4.2 Chunked LOD

Chunked LOD, or adaptations of Chunked LOD, has become the de facto standard in the computer game industry for terrain rendering [15, 11, 22]. This is because it has great scalability and much better screen-space error guarantees than Geometry Clipmap. Chunked LOD also intuitively supports operations such as out-of-core rendering [22].

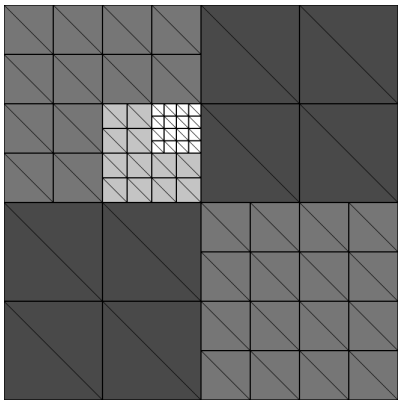


Figure 13: A quadtree with 4 levels

Chunked LOD was first introduced by Ulrich in his 2002 Siggraph talk and is built on a quadtree data structure, where each node represents what is referred to as a "chunk" [21]. Chunks are part of the terrain, the root chunk, i.e., the chunk represented by the root node, contains the whole terrain, and each child of the root chunk in turn represents a quarter of the root chunk. Every node also holds a payload with information about the given chunk, depending on the implementation of Chunked LOD the payloads content may differ, but usually it contains texture and terrain info about the chunk, as well as the maximum geometric error [21].

At run-time, the quadtree is traversed and all chunks that are outside a certain screen-space threshold and have all four children loaded to memory, gets subdivided into four smaller chunks. If the chunk does not get subdivided the chunk is rendered. This can be implemented quite efficiently completely on the GPU, which gives the CPU more time to work on other systems [15].

One major problem with this algorithm is the introduction of cracks because of the different LOD level between chunks. Ulrich solves this by introducing something he calls skirts, which is vertical borders that "hang" at the edges of each chunk and are textured so that the hole from the crack is covered. Since each chunk has a discrete LOD, this method also introduces noticeable popping when switching between LOD levels, which Ulrich solves by morphing vertices between transitions [21].

3.5 Continuous rendering

To render a chunked LOD terrain, crack prevention is needed [21]. Whereas Ulrich used skirts as an effective way to hide cracks, new and improved algorithms exist that do not produce as large artifacts as skirts. Below, two types of crack-free terrain rendering algorithms are presented.

3.5.1 Continuous Distance-Dependent Level of Detail

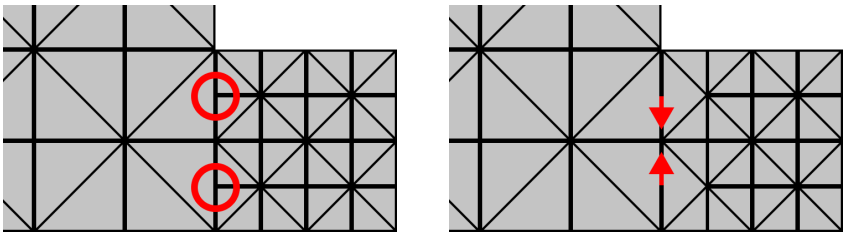
Continuous Distance-Dependent Level of Detail (**CDLOD**) was developed by Filip Strugar in 2010. Strugar based CDLOD on the ideas from both Chunked LOD and Geometry Clipmaps and aimed to overcome some of the shortcomings in both algorithms, such as stitching between different LOD levels. CDLOD also solves popping when LOD changes occur, by using a morphing method [19].

The algorithm uses a restricted quad tree, which is a quad tree where neighboring chunks cannot differ more than one level. With this restriction Strugar was able to create a morphing function that lets each LOD transition occur smoothly. The idea of the morphing function is that instead of having each chunk have a discrete LOD, each vertex has its own distance based LOD level. This makes the algorithm more akin to a CLOD type algorithm than a discrete LOD algorithm.

3.5.2 Edge patching

This algorithm has no name in the literature and will be referred to as edge patching in this thesis. It was first presented by Johan Andersson from EA, Dice in 2007 and used in their Frostbite engine. The algorithm is used to remove T-junctions, and thereby cracks, between regular grid chunks of different LOD levels. Andersson identified 9 different permutations of the chunk regular grid that would be needed to eliminate any possibilities for T-junctions to occur.

The initial algorithm could only support one level of detail between each chunk, and therefore a restricted quad tree was needed. This restriction has later been solved such that a regular quad tree can be used. Jeremy Moore from Ubisoft presented a less restricted version of this algorithm in 2018 [15].



(a) With T-junctions (b) T-junctions removed

Figure 14: Edge patching procedure

The newer version does not identify any permutations beforehand, but instead morphs the edge vertices to remove any T-junctions present in the vertex shader. This can be seen in Figure 14, where there are two T-junctions present (a). These are eliminated by moving the vertices into the closest non T-junction vertex (b) which is the middle vertex in this example.

Although this algorithm does not need a restricted quad tree, there will still be some restrictions on the chunk LOD difference. This difference is dependent on the mesh used for the chunk. A requirement for the mesh used for each chunk is that it must have an edge count of a power of 2 along the chunk border. From this, the maximum difference in LOD between chunks can be calculated by equation 1 below:

$$L_d = log_2(e) \tag{1}$$

where L_d is the LOD difference and e is the number of edges on the mesh border where e must be a multiple of two.

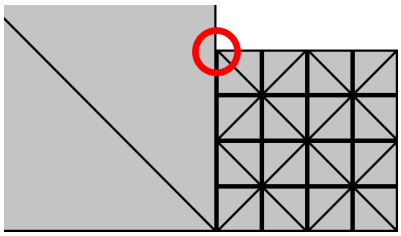


Figure 15: LOD difference of 3

In Figure 14 the chunk mesh has 4 edges along the chunk border. From Equation 1 we get that the maximum LOD difference can be 2 at most since $log_2(4) = 2$, which we can see is correct by looking at Figure 14b, since the mid vertex in (b) can still be moved. In Figure 15, we can see a LOD difference of 3, which would result in a T-junction at the marked vertex, since it cannot be moved because it is a corner of the mesh. The reason for it being a LOD difference of 3 is because the right chunk is the size of the left if it would have been halved 3 times. In Figure 14, the right chunk is only halved one time in comparison to the left chunk, hence a LOD difference of 1.

3.6 Research questions

From this survey, a few observations can be drawn, and a research question can be established. In 3.1 it is clear that some simplifications are necessary for massive terrain rendering. For this purpose, multi-resolution terrain seems like a viable option, and from 3.3.1 the conclusion can be drawn that the terrain needs to be free of T-junctions to be continuous. 3.3.2 also shows that some streaming of spatial data is needed to handle real massive terrains.

From these observations, two types of terrain LOD algorithms were identified in 3.4. Both these have pros and cons, but since Chunked LOD has better guarantees for screen-space error, it seems like the most prominent candidate. Chunked LOD is also a viable candidate to use with a CBT, which is one of the aims in this thesis.

With Chunked LOD it is possible to use both TINS and regular grids, but since TINS need to match their edge vertices to remain continuous during the whole terrain as seen in 3.3.1, regular grids are a more prominent candidate to represent each chunk.

From the Literature Survey and the thesis aims, two research questions were identified. They can be defined as:

> **What are the benefits of using a Concurrent Binary Tree in a Chunked LOD algorithm that resides completely on the GPU?**

and ...

> **How does the mesh shading pipeline compare against the traditional graphics pipeline in terrain rendering on the GPU?**

4 Methodology

In this chapter, details regarding the implementation of the different algorithms are covered. First, the Chunked LOD implementations are presented. After that, the two types of rendering solutions are described. An algorithm for rendering continuous terrain is also presented. Lastly, the methods for evaluating the implementations are explained as well as any restrictions in the implementations.

Everything presented, if it is not stated to be a part of a library or similar, are designed in this thesis.

4.1 Naive Chunked LOD

The Naive Chunked LOD system for the terrain consists of 3 steps and is based on the LOD system used in Far Cry 5. The reason why the Far Cry 5 terrain LOD system was used is because it has been implemented and proven in a real product [15]. Figure 16 shows an overview of the naive system.

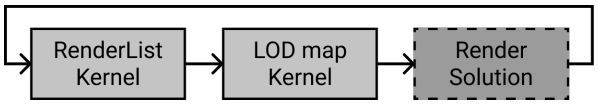


Figure 16: Naive Chunked LOD design

Renderlist Kernel - The Renderlist kernel is responsible for the traversal of the Quadtree. It does this by having 3 lists that hold node ids. The first list is the render list itself, it contains all of the nodes that are accepted by the traversal algorithm. That is, all nodes that are loaded and are within the screen space error for that frame. The other two lists, called node list and next list, are temporary storage used during the traversal algorithm seen in Algorithm 1.

Algorithm 1 Compute shader to build a renderlist. Executed on the GPU.

Require: temporary list nodeList and nextList, output list renderList

```
1: function BuildRenderList
2:     nodeId ← next node from nodeList
3:     if shouldSubdivide(nodeId) then
4:         Add nodeId's children to nextList
5:     else
6:         Add nodeId to renderList
7:     end if
8: end function
```

The Renderlist algorithm is executed by a compute shader on the GPU for each level in the tree where each entry in the list has a thread allocated to it. On line 2, the node id is extracted from the node list. This node is then evaluated on line 3 to see whether or not it can be subdivided. This involves checking if all child nodes are loaded to GPU memory, if the node is not at the bottom of the tree, and if the screen space error is above the threshold. If the node should be subdivided, all the node's children are added to the next list on line 4, otherwise the node is done and is added to the render list. After each execution, the content of node list and next list

is swapped. Before the first execution, all root tiles must be added to the node list from the CPU.

LOD map Kernel - The LOD map kernel will dispatch one thread for each sector of the terrain. The thread will then extract the LOD data of the sector from the render list and produce a texture with all LOD data for the current frame. The extraction begins with the root node of the tree, and traverses down. If a node is found in the list, it is stored as the last seen node and continues down the tree. If a null node is found, the search can be concluded and the LOD of the last seen node can be written to the LOD texture.

Algorithm 2 Naive LOD map algorithm

```
Require: renderList, lodMap
 1: function CreateLODMap(threadID)
 2:     sectorNodeId ← threadID + 4^d
 3:     lastSeenNode ← 1
 4:     currentNode ← 1
 5:     for i = d-1; i >= 0; i++ do
 6:         if currentNode ∉ renderList then
 7:             break
 8:         end if
 9:         lastSeenNode ← currentNode
10:         currentNode ← GetChildTowards(sectorNodeId)
11:     end for
12:     lodMap[sector] ← lastSeenNode.lod
13: end function
```

On line 2-4 some initial setup is done. The id of the sector that the current thread is operating on is calculated on line 2, which can be used to know how to traverse the tree on line 10. On line 6, the null node check is done. In this case it means that the current node is not a part of the render list. If it is not a part of the render list, the loop can be ended and the LOD of the last seen node can be written to the LOD map, this happens on line 12. Otherwise, the search continues and the current node is set to the last seen node on line 9.

Render Solution - The Render Solution is responsible for culling and rendering the terrain and will be presented in a separate section. But with the Naive Chunked LOD, the Render Solution will be given a render list as input.

4.2 CBT Chunked LOD

The CBT Chunked LOD is built upon the same structure as the Naive Chunked LOD. It will try to improve the Naive system by saving the state about the Quad Tree in a CBT. This means that the system does not have to iterate through all possible combinations of the tree every frame and therefore does not need a render list. Figure 17 shows an overview of the CBT Chunked LOD system.

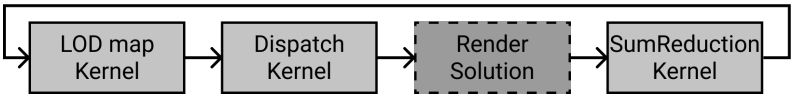


Figure 17: CBT based Chunked LOD design

LOD map Kernel - The LOD map Kernel for the CBT Chunked LOD is very similar to the LOD map kernel for the naive implementation. The main difference being the way a node is verified. With a CBT present, the algorithm has the possibility to check whether if the node is a leaf node directly. If it is, no further search is needed and the depth can be written directly.

Algorithm 3 CBT LOD map algorithm

```
Require: renderList, lodMap
function CreateLODMap(threadID)
    sectornodeId ← threadID + 4^d
    currentNode ← 1
    for i = d-1; i >= 0; i++ do
        if cbt_IsLeafNode(currentNode) then
            break
        end if
        currentNode ← GetChildTowards(sectornodeId)
    end for
    lodMap[sector] ← currentNode.lod
end function
```

Dispatch Kernel - The dispatcher kernel is from the CBT library. It consists of only one thread, this thread reads the amount of leaf nodes from the CBT and writes it into an indirect arguments buffer.

Render Solution - The Render Solution is responsible for culling and rendering the terrain and will be presented in a separate section. But with the CBT Chunked LOD, the Render Solution will be given the CBT as input.

Sum-reduction Kernel - The Sum-reduction Kernel is from the CBT library. It updates the CBT heap after any potential merges or splits has occurred in the Render Solution.

4.2.1 CBT as quadtree

By default, a CBT only represents a binary tree. With some adaption, however, it is possible to represent a quadtree using a binary tree, hence it is possible to represent a quadtree with a CBT.

To represent a quadtree of depth d, a CBT of depth $2 \times d$ is required. The split and merge operations will be modified to split and merge two levels of nodes in each operation. A wrapper function for split and merge, called QuadMerge and QuadSplit can be seen in Algorithm 4 and 5.

Algorithm 4 Merges a CBT node such that the CBT keeps being a quadtree

```
Require: Must be executed on all 4 children
function QuadMerge(cbt: CBT, node: CBT_Node)
    MergeNode(cbt, node)
    parent ← GetParent(cbt, node)
    MergeNode(cbt, parent)
end function
```

Algorithm 5 Splits a CBT node such that the CBT keeps being a quadtree

```
1: function QuadSplit(cbt: CBT, node: CBT_Node)
2:     SplitNode(cbt, node)
3:     leftChild ← GetLeftChild(cbt, node)
4:     rightChild ← GetRightChild(cbt, node)
5:     SplitNode(cbt, leftChild)
6:     SplitNode(cbt, rightChild)
7: end function
```

4.3 Render Solution

There will be two rendering solutions presented, one using a Compute shader to cull the landscape and render it using a vertex shader with instanced rendering, we will refer to this solution as the Instance Render Solution (**IRS**). The other solution will use only the mesh shading pipeline. The culling of the terrain will occur on a task shader with 32 threads in each workgroup, since that is the size of the waves on the hardware available. Each task shader will dispatch some mesh shaders, where each mesh shader work group will render one tile each. We will refer to this solution as the Mesh Shading Solution (**MSS**).

Both Render Solutions will use view frustum culling as the culling algorithm. The view frustum culling algorithm will use spherical bounds for each tile. This is an easy to implement algorithm with low run-time cost, since only a radius from the center of the tile is compared to the view frustum. However, spherical bounds may result in some tiles being completely outside the view frustum still being rendered.

The input to the culling will depend on the Chunked LOD algorithm in use. For the Naive implementation, the number of threads must be equal to the size of the render list, since the render list is not a packed datastructure. For the CBT Chunked LOD, the right number of threads are dispatched due to the dispatch kernel, but each thread must decode the right node from the CBT before any culling can occur.

4.3.1 MSS Culling

The algorithm described in Algorithm 6 is used to cull terrain tiles with the Mesh Shader Solution and is executed on a task shader.

Algorithm 6 View frustum culling in task shader

```
Require: nodeList = list of nodes
1:  function CullTiles(GlobalThreadID)
2:      node ← nodeList[GlobalThreadID]
3:      if node ≠ NULL then
4:          if inViewFrustum(node.pos) then
5:              hasRenderTarget ← true
6:          end if
7:      end if
8:      offset ← WavePrefixCountBits(hasRenderTarget)
9:      tasks ← WaveActiveCountBits(hasRenderTarget)
10:     if hasRenderTarget then
11:         Payload[offset] ← node
12:     end if
13:     DispatchMesh(tasks, Payload)
14: end function
```

If a tile is within the view frustum of the camera, the thread is said to have a render target. This is evaluated on line 2 to 7. On line 8 and 9 two different wave intrinsic functions are used. "WavePrefixCountBits" will count how many threads in the workgroup with an id lower than the current thread has a render target, i.e., gets an offset to store the tile information in the payload. This offset is crucial so that no thread overwrites data from another thread since the payload is shared between all threads in a workgroup. The "WaveActiveCountBits" counts the total amount of threads in the workgroup that has a render target. Each tile will be dispatched with a certain payload, if the thread has a node with a render target, the node's render target is added to the payload as seen on line 10 to 12. Lastly, on line 13, the task shader dispatch *tasks* number of mesh shaders with the payload.

4.3.2 IRS Culling

Algorithm 7 is used by the Instance Render Solution. It does the same culling as the MSS algorithm, the only difference is that Algorithm 7 is running on a compute shader and therefore does not dispatch tiles for rendering.

Algorithm 7 View frustum culling in compute shader

```
Require: nodeList = list of nodes
1:  function CullTiles(GlobalThreadID)
2:      node ← nodeList[GlobalThreadID]
3:      if node ≠ NULL then
4:          if inViewFrustum(node) then
5:              instancesToDraw ← (atomic) instanceToDraw + 1
6:              instanceArgs ← node
7:          end if
8:      end if
9:  end function
```

Because compute shaders cannot dispatch for rendering, the algorithm packs all tiles that are not culled into a buffer. This buffer can later be used as arguments for indirect drawing. Since all threads want to write to the arguments buffer, a data lock is needed to avoid race conditions. This can be seen on line 5-6, where if a node is

within the view frustum, the value of instancesToDraw is atomically increased and the node's render data is appended to the list of instance arguments.

4.3.3 Continuous rendering

To visualize the terrain, a combination of an adaptation of CDLOD and edge patching was used. The CDLOD algorithm was used to create vertex morphing when LOD levels were changing to mitigate popping. It was combined with edge patching to support a greater span of LOD levels between neighboring tiles. The algorithm is described in Algorithm 8. While this algorithm is made to be crack-free, it cannot be guaranteed. To guarantee that the terrain will be free of cracks, the tiles must have a resolution such that equation 1 gives a span equal or greater than the depth of the tree. Both the MSS and the IRS will use this continuous render algorithm, MSS will run it in a Mesh Shader while IRS will use a Vertex Shader.

Algorithm 8 Algorithm for offsetting a vertex

Require: $tilePayload \leftarrow position, edgeSize, lodDiff, morphValue$
Ensure: $y = x^n$

```
1:  function OffsetVertex(vertex)
2:      offset ← (0,0,0)
3:
4:      if vertex.z = VertexMax then                          ▷ At north border
5:          offset.x ← -edgeSize * (vertex.z mod 2^lodDiff.north)
6:      else if vertex.z = 0 then                             ▷ At south border
7:          offset.x ← -edgeSize * (vertex.z mod 2^lodDiff.south)
8:      else
9:          if vertex.x mod 2 then
10:             offsetDirection ← 2 × ((vertex.x/2)mod2) − 1
11:             offset.x ← offsetDirection × edgeSize × morphValue
12:         end if
13:     end if
14:
15:     if vertex.x = VertexMax then                          ▷ At east border
16:         offset.z ← -edgeSize * (vertex.x mod 2^lodDiff.east)
17:     else if vertex.x = 0 then                             ▷ At west border
18:         offset.z ← -edgeSize * (vertex.x mod 2^lodDiff.west)
19:     else
20:         if vertex.z mod 2 then
21:             offsetDirection ← 2 × ((vertex.z/2)mod2) − 1
22:             offset.z ← offsetDirection × edgeSize × morphValue
23:         end if
24:     end if
25:
26:     return vertex + offset
27: end function
```

The algorithm will run for each vertex in all tiles, and the goal is to offset the vertex to match the morphing and edge patching. On line 4-13 the x translation of the vertex is managed. Line 4 evaluates whether the vertex is on the north border of the tile. If the vertex is a border vertex, the vertex is offset depending on the LOD difference between the north neighbor. The same happens on line 6-7, except for the south border. If the vertex is not a border vertex, the vertex is morphed according

to the morph value, this can be seen on line 9-12.

On line 15-24, the same procedure as on line 4-13 is executed, apart from it being the z translation that is offset with regards to the west and east border, and the morph value.

4.4 Evaluation methods

This section presents the different methods selected to evaluate the performance of the implementations. Three different methods have been identified.

4.4.1 Fragment shader load

Since meshes usually have more visual properties than just a single color or texture, this may have to be accounted for. This is work that is done in the fragment shader stage and is therefore executed for every fragment. Between the two pipelines, the fragment shading should be identical; but to verify that the fragment shader does not impact any pipeline differently, this is tested. A few static positions will be chosen, and during multiple frames performance will be measured with both the Naive Chunked LOD and the CBT Chunked LOD. Each time, the work done on the fragment shader will increase linearly, starting from 0 texture samplings and ending at 128.

This experiment would have been crucial if per-primitive occlusion culling would have been implemented on the mesh shading pipeline. But due to time limitations, this lies outside the scope of this thesis.

4.4.2 Chunked LOD evaluations

This experiment will evaluate CBT Chunked LOD and the Naive Chunked LOD algorithm performances as the depth of the tree increases. To make sure that the render solution will not interfere with the result, only the compute shader version will be tested for both versions since it is possible to measure separate from the rendering. On the Naive Chunked LOD algorithm, this experiment will take into account the building of the render list and the filling of the LOD map. For the CBT Chunked LOD we will measure the LOD map creation, dispatch kernel and the sum reduction kernel.

4.4.3 Complete terrain

This experiment aims to evaluate the complete system in a real game environment; albeit lacking any game mechanics. A fragment shader that runs 32 texture samples will be used, this is selected arbitrarily, but should reflect the amount of texture samples a terrain would do according to internal data from Paradox. The experiment will run on three different maps, one which is noticeable smooth therefore having a low geometric error, one map that can be seen as an average terrain, and the last with an exceptionally uneven terrain that can be seen as a worst-case scenario. Both Chunked LOD algorithms will be evaluated with both the Mesh Shading Solution and the Instance Render Solution.

4.5 Implementation Restrictions

Because of the scope of the thesis, some restrictions had to be made.

1. Each tile will be statically loaded on launch (No Out-of-core rendering).
2. No texture streaming will be implemented, all data will be loaded to the GPU on launch.
3. No texture difference between LOD levels will be used.

5 Results

This chapter presents the results of the experiments outlined in the previous chapter.

5.1 Terrain properties

The terrain uses a heightmap of size 4096x4096. The least amount of subdivision possible will be four-by-four, i.e., there will always be at least four-by-four tiles loaded. One way to describe this is by having a value called root depth, meaning the depth of the quadtree that should be considered the root. In this case the root depth will be 2 since $4^2 = 16$. The terrain will at maximum be split up into 1024×1024 tiles. The tiles at the lowest level of the quad tree will be referred to as sectors. A sector will therefore have the depth of eight in the quadtree from the root depth.

The maximum error of the terrain will be restricted to a maximum of four pixels. This was chosen arbitrarily but should represent a common error threshold used in other applications.

Variable name	Value	Description
$Root_D$	2	Root depth
$Tree_D$	8	Depth from root depth
$Error_P$	4	Max pixel error

Table 3: Global properties of the terrain

Depending on the implementation of the Chunked LOD algorithm, the terrain can be stored in a packed or sparse array.

5.2 Test computer hardware

The hardware on which the experiments were executed can be seen in Table 4. The graphics API used for the tests were DirectX12.

CPU	Intel Core i7-8700K @ 3.7GHz, 6 cores
RAM	16GB 2666MHz
GPU	Nvidia GeForce RTX 2060

Table 4: Hardware of the test computer machine

Everything was implemented using the Clausewitz Engine, which is Paradox Interactives proprietary game engine. The computer used were running Windows 10, 21H2.

5.3 Fragment shader load results

Comparison of MSS (Mesh Shading Solution) and IRS (Instance Render Solution) when fragment shader load is increased.

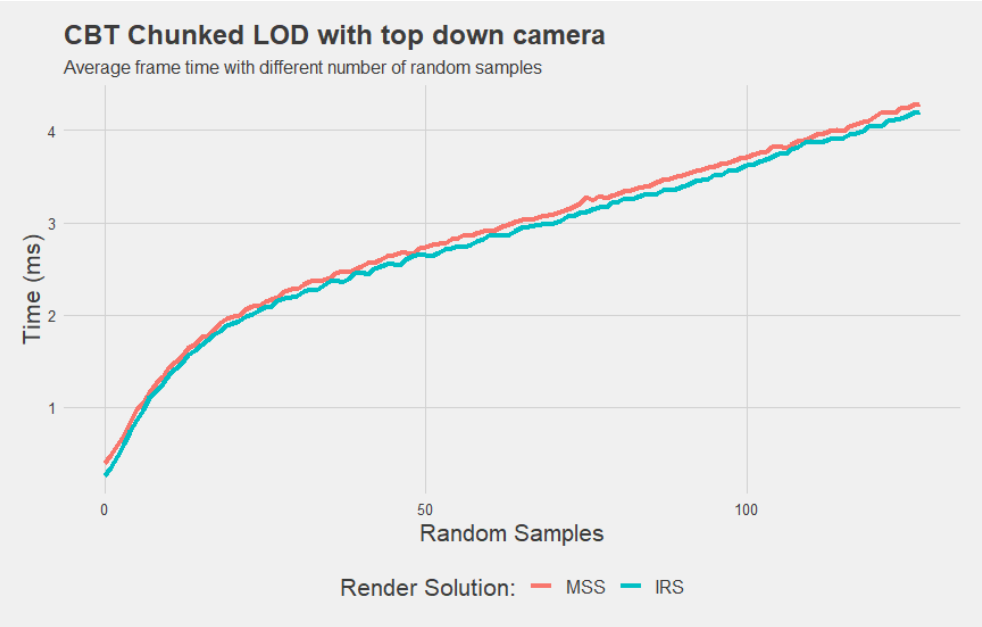


Figure 18: Results of fragment shader test with CBT Chunked LOD and a top down camera.

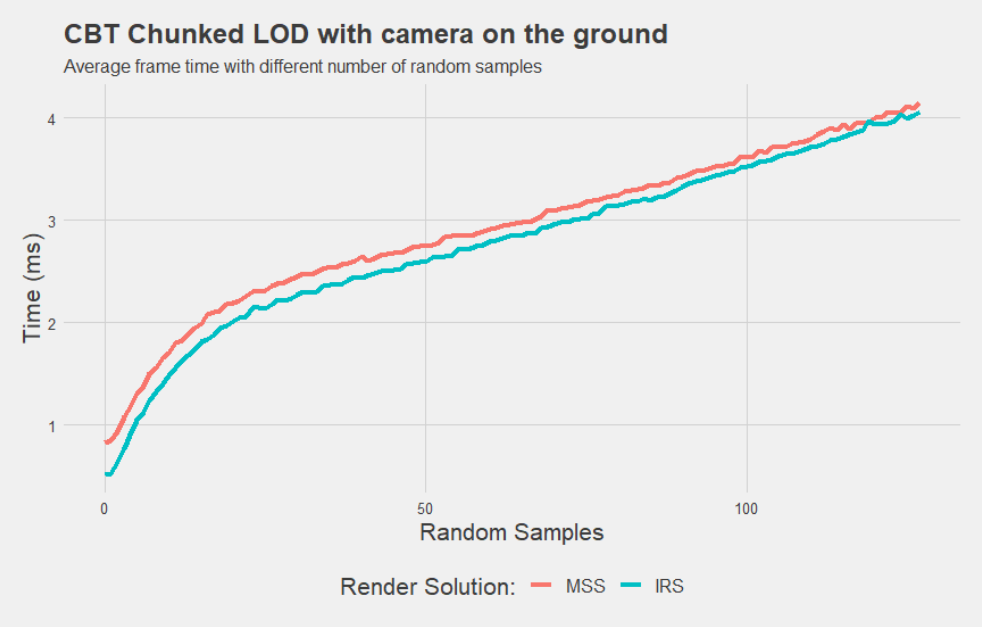


Figure 19: Results of fragment shader test with CBT Chunked LOD and a ground camera.

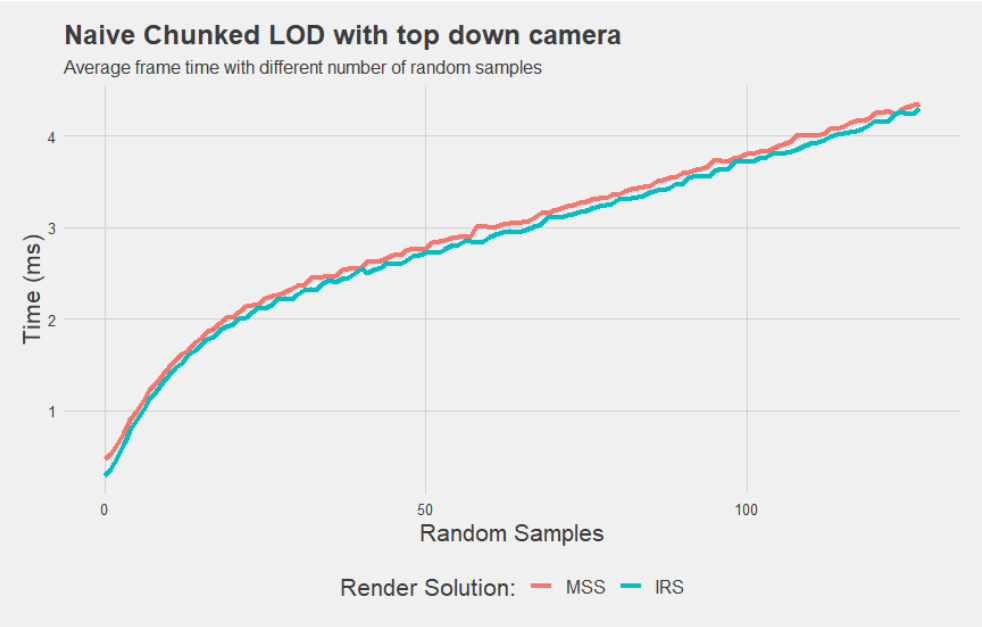


Figure 20: Results of fragment shader test with Naive Chunked LOD and a top down camera.

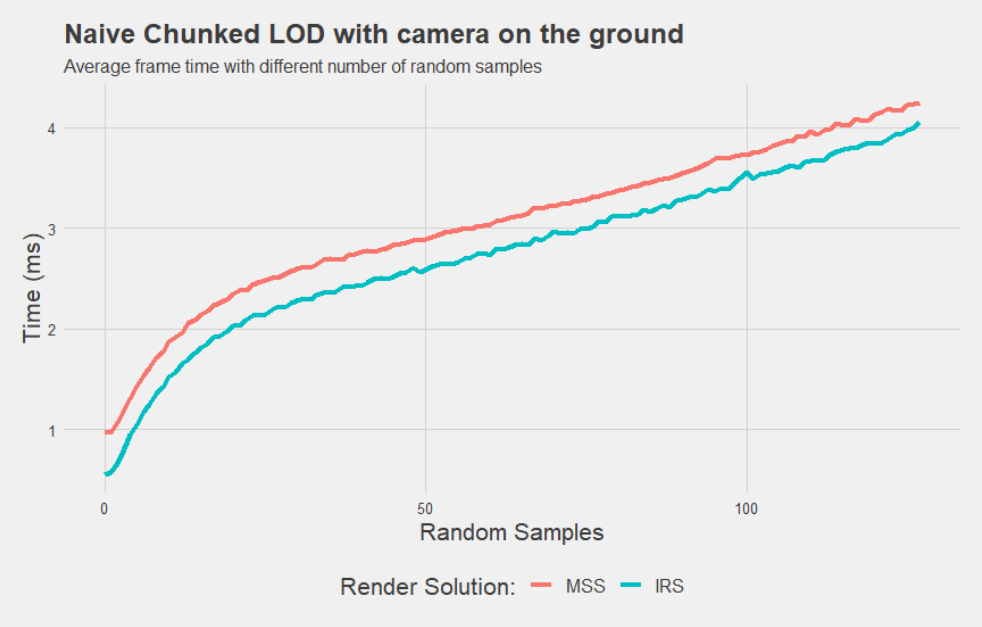


Figure 21: Results of fragment shader test with Naive Chunked LOD and ground camera.

The results of the fragment shader load came out as expected. The render solutions did not behave differently when the number of texture samples rose. It can also be observed that the performance of the solutions did not differ by any significant amount when using a CBT Chunked LOD algorithm. This is probably due to the fact that the CBT Chunked LOD presents all tiles to be rendered in a packed order, while the Naive Chunked LOD is unpacked.

5.4 Chunked LOD results

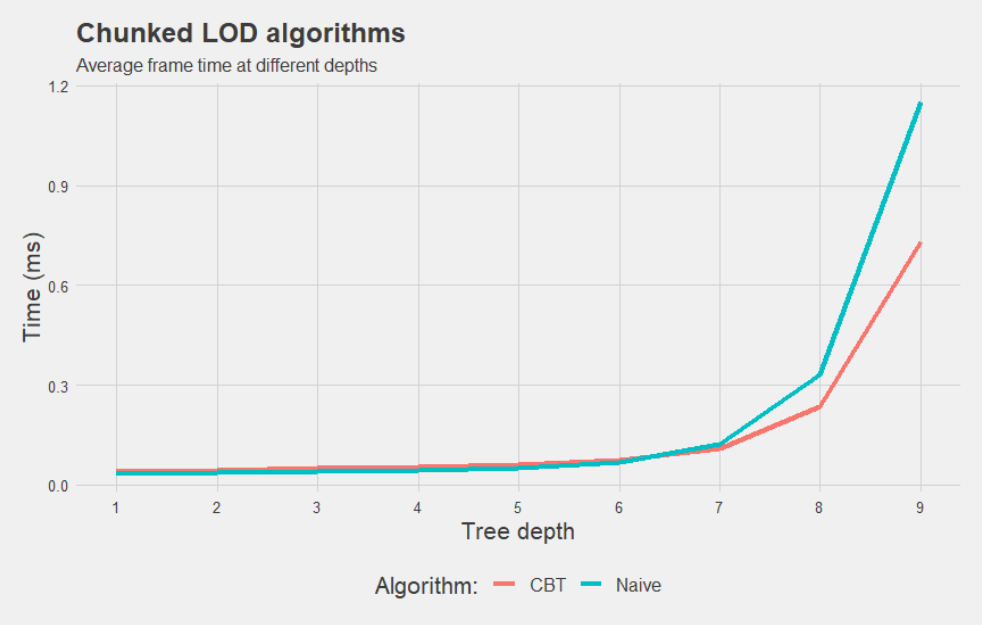


Figure 22: Result of different LOD implementations on varying depth.

Figure 22 shows the results from the Chunked LOD implementation evaluations. The test where run on different depth, but all with a root depth of 2. It can be seen that both algorithms fair very similar up to a depth of 7 (with a total depth of 9). After that, the strengths of the CBT implementations starts to become visible. However, at this depth the size of the CBT data-structure starts to become a thing of concern. At the depth of 9 (total depth 11), the size of the CBT data structure is around 16,8Mb. The next level would have been around 60Mb, but this was too large to allocate as a single data structure on the GPU. From this it can be concluded that, while the CBT do yield an improvement in terms of runtime, it is impractical for any real use.

5.5 Complete terrain results

The complete terrain was split up in three subtests, as mentioned in the method. A smooth, an average and an uneven terrain was all evaluated.

5.5.1 Smooth terrain

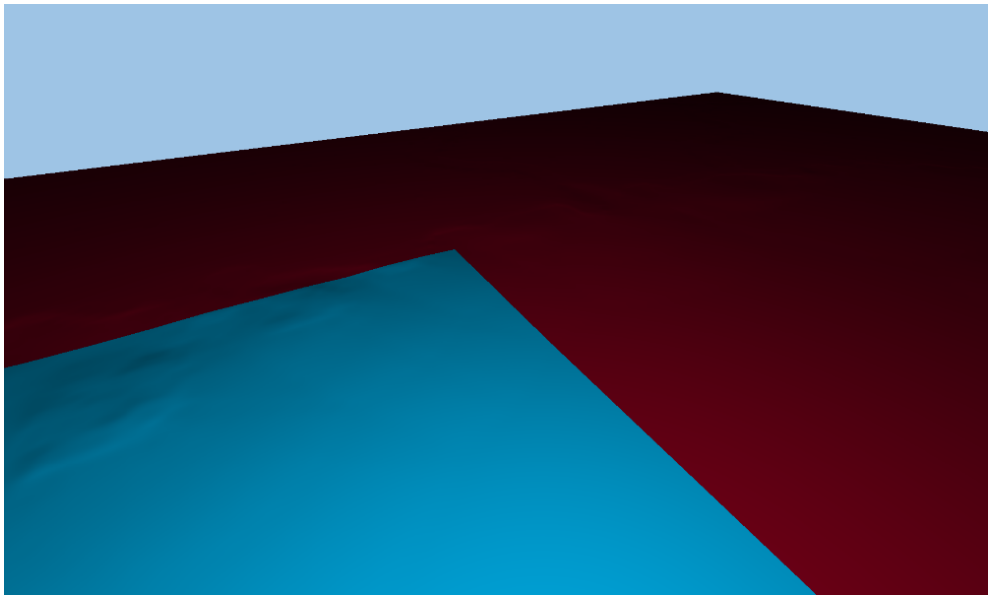


Figure 23: Rendering of the smooth terrain

Figure 24: Heightmap of the smooth terrain

The smooth terrain is an example of a "best" case scenario. This is where the whole terrain has an exceptionally low pixel error, such that a low-resolution tile still represents the terrain accurately. In practice, this means that the tiles will not subdivide as much since the error produced by each tile will be low. In Figure 23 the terrain

can be seen, which is rendered from the height map in Figure 24. The height map is a scan of the Bothnian Bay.

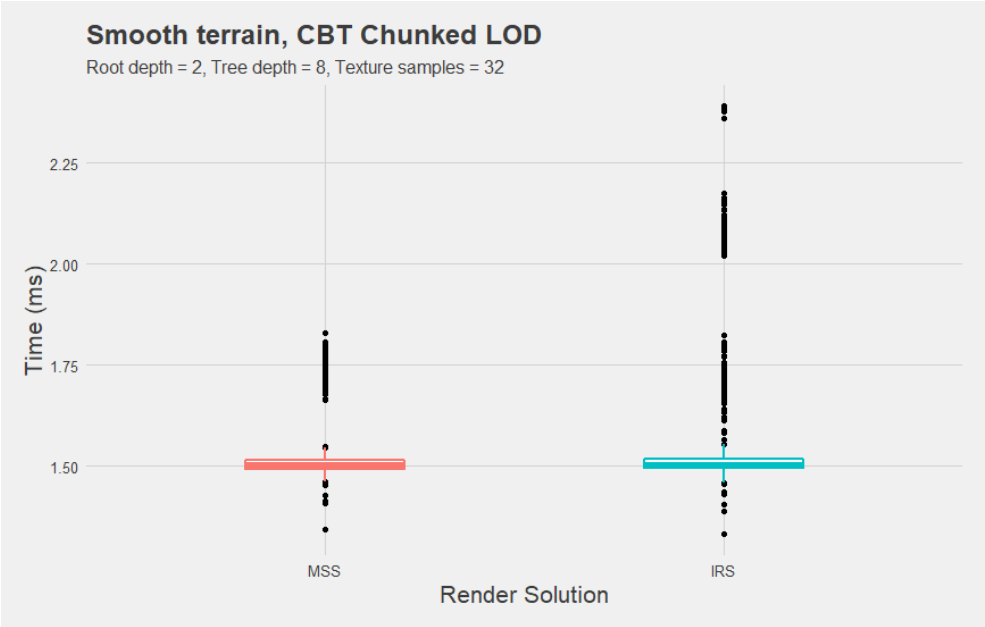


Figure 25: Results from CBT Chunked LOD terrain with a smooth map

When the terrain is smooth, it can be seen that both algorithms are very equal in terms of performance. IRS seems to be somewhat slower and more unstable due to more outliers.

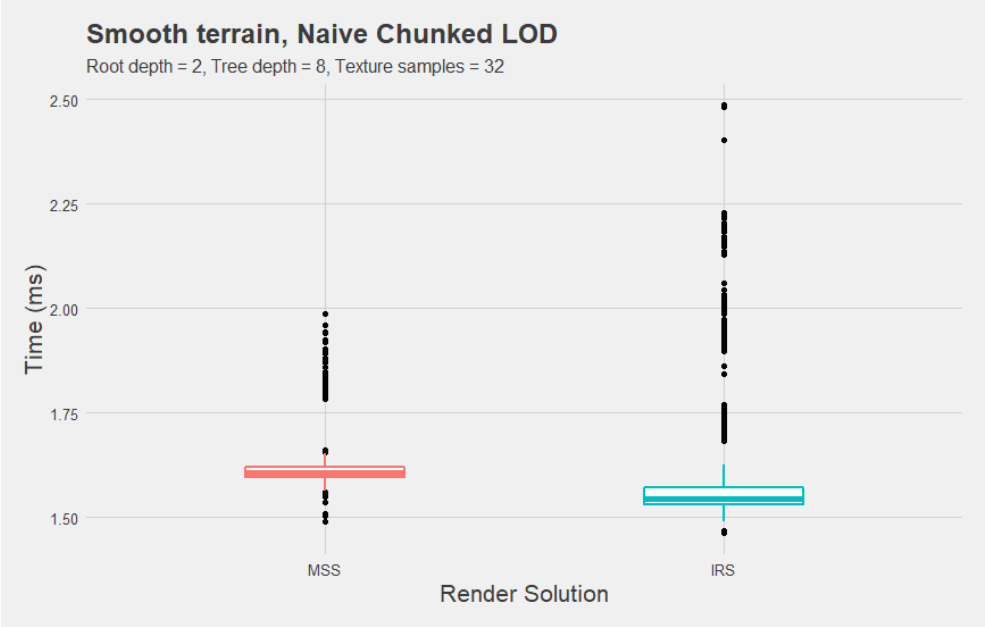


Figure 26: Results from Naive Chunked LOD terrain with a smooth map

When the LOD algorithm is not using a CBT, the MSS performs arount 0.1ms worse

than IRS, but seems to be more stable than IRS. With a CBT Chunked LOD, the results show that with a smooth terrain, both render solutions can be considered equals as there is no significant performance loss with either. We can also observe that the CBT Chunked LOD performs better no matter the render solution, which is expected.

5.5.2 Average terrain

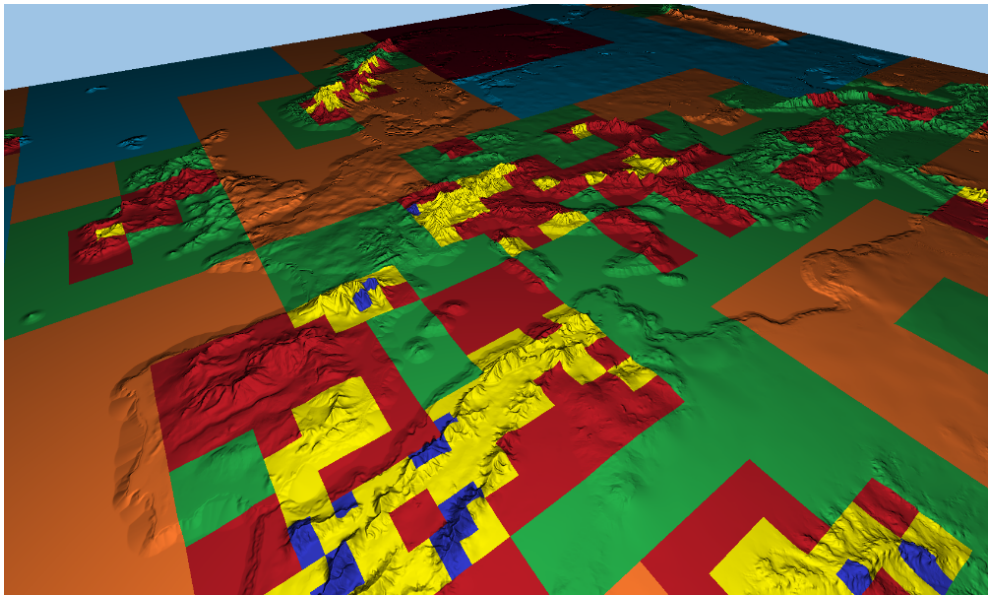


Figure 27: Rendering of the average terrain

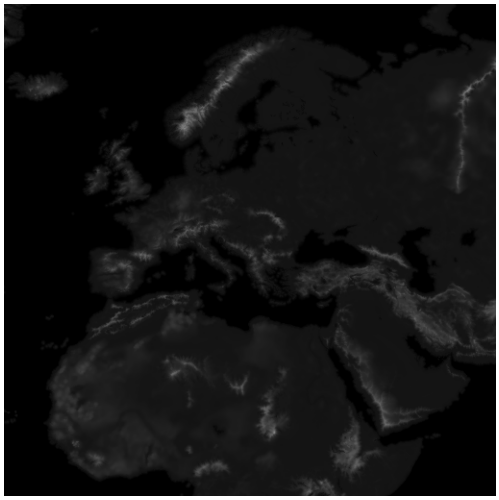


Figure 28: Heightmap of the average terrain

An "average" heightmap is a heightmap that is supposed to represent a typical terrain. It includes both mountains and flat areas. In this case a heightmap of Europe and northern Africa was used. The average terrain can be seen in Figure 27 and is generated by the heightmap in Figure 28.

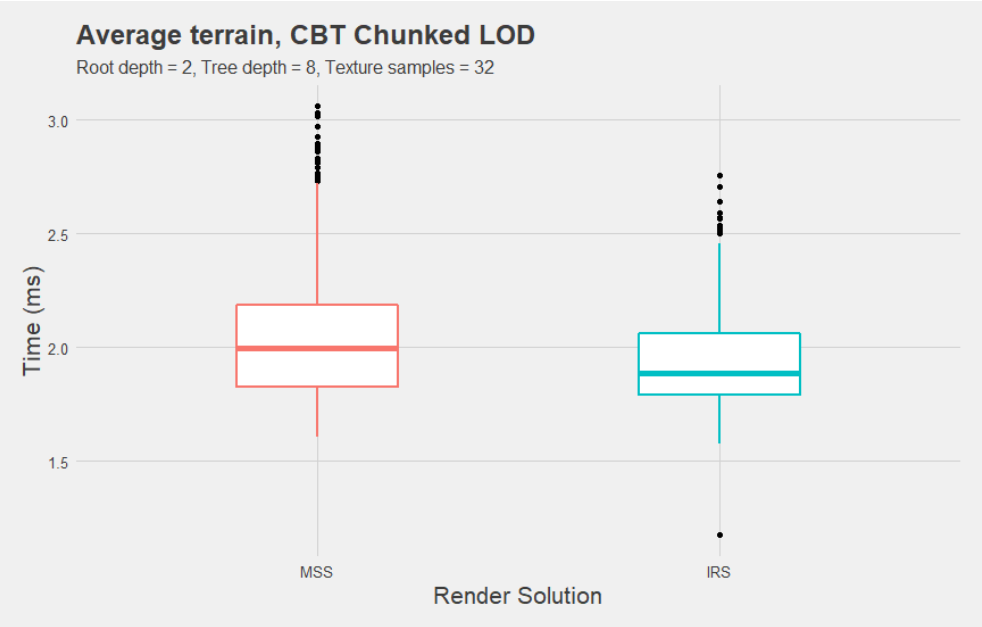


Figure 29: Results from CBT Chunked LOD terrain with an average map

By Figure 29 we can observe that the mean is slightly higher with MSS, but that both render solutions are very similar. In contrast to the smooth terrain, it seems like it is the MSS this time that is the more unstable render solution when observing the span of the box and the outliers.

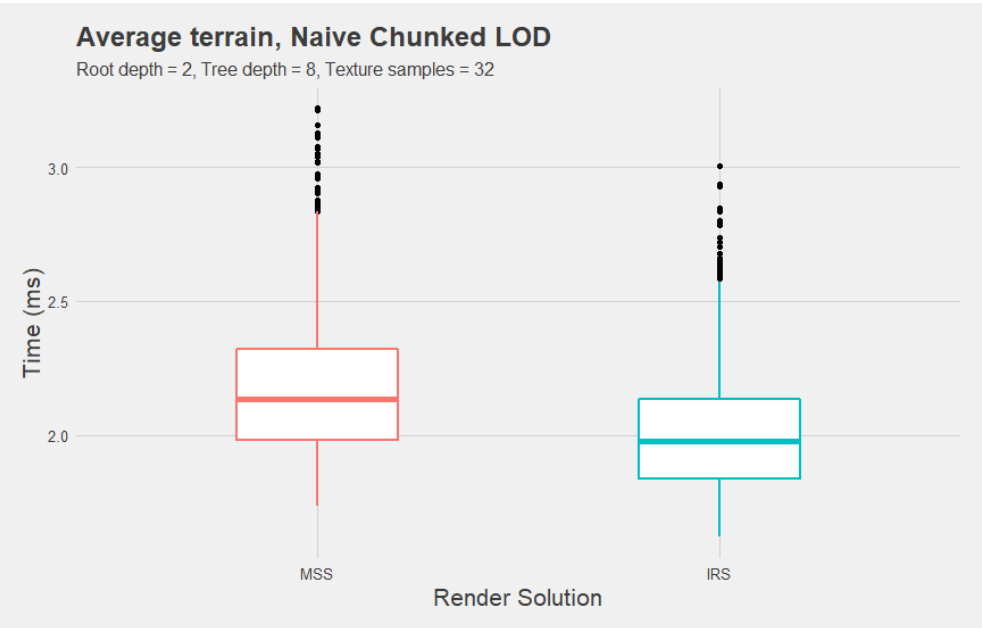


Figure 30: Results from Naive Chunked LOD terrain with an average map

In Figure 30 it can be seen that, once again, the CBT is a improvement from the naive algorithm. It also follows the same pattern as Figure 29, that the MSS is the more unstable and slight slower render solution.

5.5.3 Uneven terrain

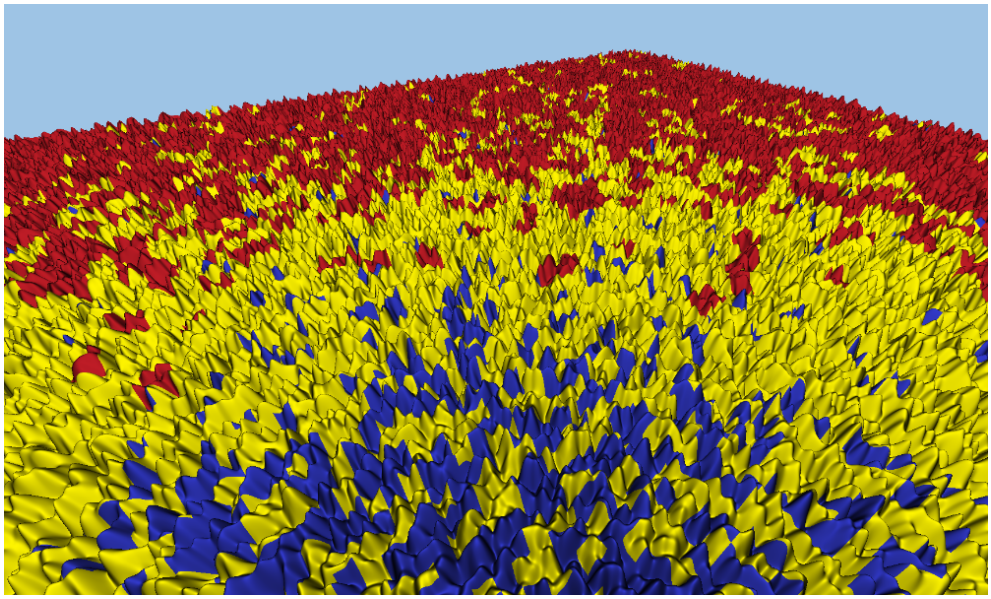


Figure 31: Rendering of the uneven terrain

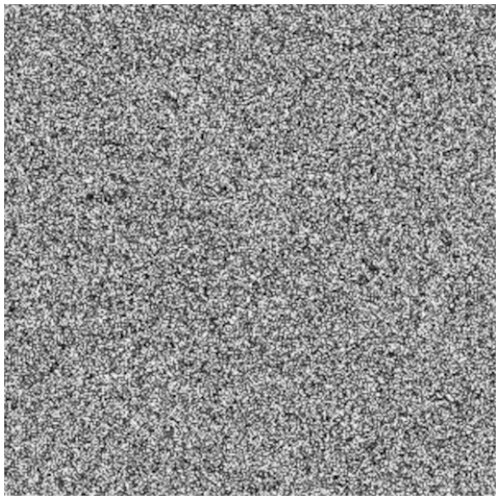


Figure 32: Heightmap of the uneven terrain

The last terrain to be evaluated is very uneven, to the extent that it would be highly unlikely to see this kind of terrain in a real application. This terrain represents a worst-case scenario, which means that the terrain will have to be very subdivided to maintain the visual quality. The rendered terrain can be seen in Figure 31 and the height map in Figure 32.

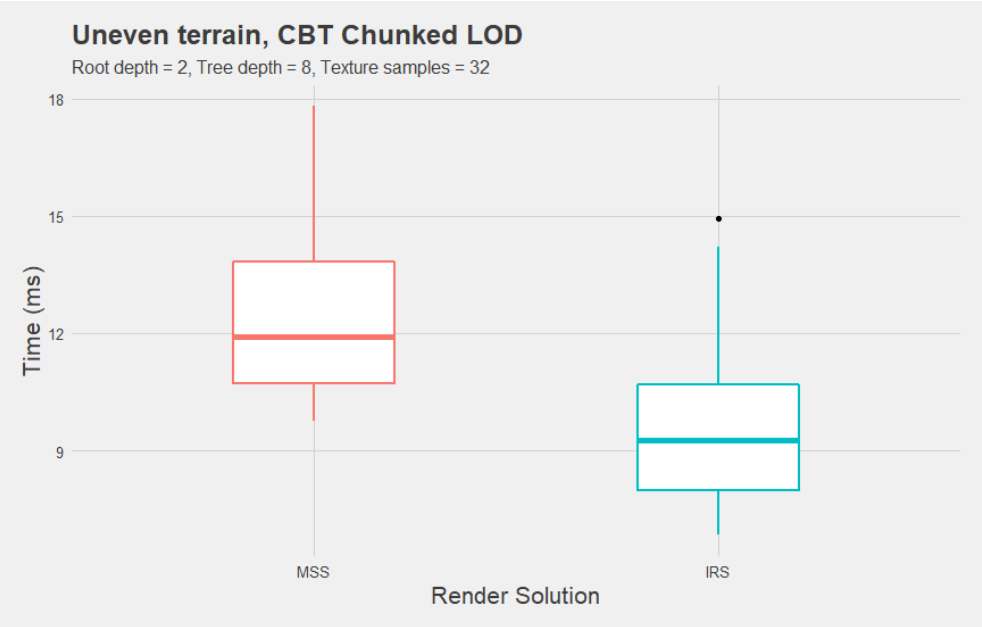


Figure 33: Results from CBT Chunked LOD terrain with an uneven map

The rendering solution seems to have a much bigger impact on a very uneven terrain. Both the mean values and overall performance is much better with IRS.

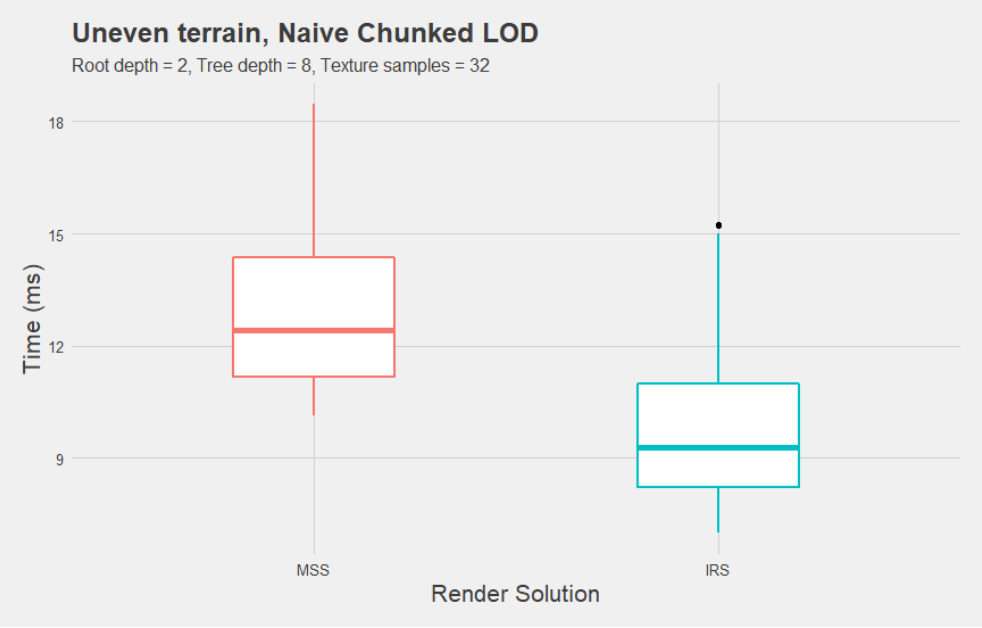


Figure 34: Results from Naive Chunked LOD terrain with an uneven map

With the Naive Chunked LOD algorithm, there is a very slight increase in frame times with the MSS, while the IRS is almost the same as with CBT Chunked LOD. This is probably due to the unpacked nodes of the Naive algorithm, which seems to be why the MSS performs worse.

6 Analysis

In this chapter, the results from the previous chapter is analyzed and evaluated in terms of performance. I will also discuss the usefulness of the different methods.

6.1 CBT Performance

By observing the results from the Chunked LOD comparison in Section 5.4, we can see that the CBT datatype can in fact improve the performance of a Chunked LOD system. However, we can also observe that, while there is an improvement, it starts to be noticeable first at a depth of 8 (total depth 10). Also, when comparing these results against the fragment shader load experiment, it is obvious that the 0.4ms gained on a tree of depth 9 fades in comparison to the work of the fragment shader. This is especially noticeable if you have an uneven terrain and a lot of texture samples as seen in the uneven terrain experiment (Subsection 5.5.3). Therefore, the answer the research question "What are the benefits of using a Concurrent Binary Tree in a Chunked LOD algorithm that resides completely on the GPU?" is that it can improve the performance of a Chunked LOD algorithm but in the general case not by a significant amount.

6.2 Render Solution Performance

With both an average and uneven terrain, the IRS beat mesh shading with an observable margin. This might not come as a surprise, however, since the graphics pipeline is very efficient when rendering instanced meshes. Surprisingly, however, is the fact that IRS must cull and pack all tiles before rendering but still manage to perform better than mesh shading. As I see it, there can be two reasons why this is.

1. Instanced rendering is significantly faster than mesh shading rendering, due to hardware optimizations. Therefore, the time taken to pack the terrain is less than the time gained when doing the actual rendering.
2. Each task shader must allocate memory for its payload each time it is dispatched. This allocation is taking more time than the packing into a preallocated array procedure.

I would argue that it is a combination of these two that makes the IRS solution better suited in a terrain render environment. This is based on the fact that MSS is always performing better with the CBT Chunked LOD, where the tiles are packed together. This causes more of the payloads to be filled more, since with the Naive Chunked LODs unpacked tiles it could easily be payloads that are almost empty. Also, since the payload size is constant, almost empty payloads waste a lot of bandwidth.

To answer the research question "How does the mesh shading pipeline compare against the traditional graphics pipeline in terrain rendering on the GPU?", the traditional pipeline is overall faster than the mesh shading pipeline for terrain rendering and is more widespread on all graphics cards because of its age. With that said, the mesh shading pipeline did not perform significantly worse and should therefore not be discarded as it can be a viable option, although at this stage more research is required.

6.3 Usefulness discussion

One conclusion that can be drawn regarding usefulness is that, even though the MSS was performing worse than IRS in most cases, it did not perform significantly worse. This opens the door for new rendering algorithms that can use adaptive number of vertices in each tile, such as TIN networks. It also makes it possible to try and utilize the strength of mesh shading, such as per-primitive culling. This could be especially useful when a terrain has a very heavy fragment shader, so that no unnecessary fragment shaders are executed.

At the current state, I would argue against using the mesh shading pipeline for terrain rendering due to two reasons. First, mesh shading is slower than instance rendering, as seen from the results between IRS and MSS. Secondly, it is a very new pipeline, which was introduced with the RTX 20-series of graphics cards. Therefore, the number of people that can utilize this pipeline might not be that high.

7 Conclusion

This thesis has presented two GPU implementations of a Chunked LOD algorithm, one being naive in the selection of tiles while the other utilizes a Concurrent Binary Tree. The thesis has also presented an algorithm for rendering Chunked LOD tiles continuously.

The use of a CBT can help increase the performance of a Chunked LOD algorithm. However, this is very situational and in most practical cases, it will not increase the performance by any significant amount. If the algorithm of choice is GPU based, it should be fine. So, the answer the research question "What are the benefits of using a Concurrent Binary Tree in a Chunked LOD algorithm that resides completely on the GPU?" is, it can improve the performance of a Chunked LOD algorithm, but in the general case not by a significant amount.

In terms of performance, there is no doubt that the old graphics pipeline is more than well suited for current generation of terrain rendering algorithms. To answer the research question "How does the mesh shading pipeline compare against the traditional graphics pipeline in terrain rendering on the GPU?", the traditional pipeline is overall faster than the mesh shading pipeline for terrain rendering and is more widespread on all graphics cards because of its age. With that said, the mesh shading pipeline did not perform significantly worse and should therefore not be discarded as it can be a viable option, although at this stage more research is required.

7.1 Further work

One topic for future work identified during this Thesis is the use of more advanced Culling algorithms with Mesh Shader. Such as occlusion culling on both a per tile basis and per tile with per primitive basis. Since it is only possible to do a per tile culling with compute shaders, while mesh shaders allow for per primitive. This opens the possibility that, while the mesh shading pipeline was slower in general, it might be faster to use a mesh shading with advanced culling if the terrain has a high amount of texture samples.

References

[1] Tomas Akenine-Möller et al. *Real-Time Rendering 4th Edition*. Boca Raton, FL, USA: A K Peters/CRC Press, 2018, p. 1200. ISBN: 978-1-13862-700-0.

[2] Emil Bertilsson. "Dynamic Creation of Multi-resolution Triangulated Irregular Network". 2015.

[3] Patrick Cozzi and Kevin Ring. *3D Engine Design for Virtual Globes*. 1st. http://www.virtualglobebook.com. CRC Press, June 2011. ISBN: 978-1568817118.

[4] Leila De Floriani, Leif Kobbelt, and Enrico Puppo. "A Survey on Data Structures for Level-of-Detail Models". In: 2005, pp. 49–74. ISBN: 978-3-540-21462-5.

[5] M. Duchaineau et al. "ROAMing terrain: Real-time Optimally Adapting Meshes". In: *Proceedings. Visualization '97 (Cat. No. 97CB36155)*. 1997, pp. 81–88.

[6] Jonathan Dupuy. "Concurrent Binary Trees (with Application to Longest Edge Bisection)". In: *Proc. ACM Comput. Graph. Interact. Tech.* 3.2 (Aug. 2020).

[7] Michael Garl and Paul Heckbert. "Fast Polygonal Approximation of Terrains and Height Fields". In: (1995).

[8] Shawn Hargreaves. *Reinventing the Geometry Pipeline: Mesh Shaders in DirectX 12*. 2020. URL: https://www.youtube.com/watch?v=CFXKTXtil34.

[9] Sarah Jobalia and Microsoft. *Coming to DirectX 12— Mesh Shaders and Amplification Shaders: Reinventing the Geometry Pipeline*. 2019. URL: https://devblogs.microsoft.com/directx/coming-to-directx-12-mesh-shaders-and-amplification-shaders-reinventing-the-geometry-pipeline/.

[10] Christoph Kubisch. *Introduction to Turing Mesh Shaders*. 2021. URL: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/.

[11] Alen Ladavac and Croteam. "Advanced Graphics Techniques Tutorial: Four Million Acres, Seriously: GPU-Based Procedural Terrains in 'Serious Sam 4: Planet Badass'". In: (Game Developers Conference). 2019. URL: https://www.gdcvault.com/play/1026349/Advanced-Graphics-Techniques-Tutorial-Four.

[12] Shi Li et al. "Multi-resolution terrain rendering using summed-area tables". In: *Computers & Graphics* 95 (2021), pp. 130–140. ISSN: 0097-8493.

[13] Frank Losasso and Hugues Hoppe. "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids". In: *ACM Trans. Graph.* 23.3 (2004), pp. 769–776. ISSN: 0730-0301.

[14] David Luebke et al. *Level of Detail for 3D Graphics*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 9780080510118.

[15] Jeremy Moore and Ubisoft. "Terrain Rendering in 'Far Cry 5'". In: (Game Developers Conference). 2018. URL: https://www.gdcvault.com/play/1025480/Terrain-Rendering-in-Far-Cry.

[16] Renato Pajarola and Enrico Gobbetti. "Survey of semi-regular multiresolution models for interactive terrain rendering". In: *The Visual Computer* 23 (2007), pp. 583–605.

[17] Jon Peddie. "Mesh Shaders Release the Intrinsic Power of a GPU". In: (ACM Siggraph). 2021. URL: https://blog.siggraph.org/2021/04/mesh-shaders-release-the-intrinsic-power-of-a-gpu.html/.

[18] Josh Petty. *What is a Polygon Mesh?* URL: https://conceptartempire.com/polygon-mesh/.

[19] Filip Stugar. "Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD)". In: (2010). URL: https://github.com/fstrugar/CDLOD.

[20] Unity Technologies. "Experimenting with Concurrent Binary Trees for Large Scale Terrain Rendering". In: 2021. URL: https://www.youtube.com/watch?v=0TzgFwDmbGg.

[21] Thatcher Ulrich. "Rendering Massive Terrains using Chunked Level of Detail Control". In: (SIGGRAPH). 2002. URL: http://tulrich.com/geekstuff/sig-notes.pdf.

[22] Mattias Widmark and EA Dice. "Terrain in Battlefield 3: A Modern, Complete and Scalable System". In: (Game Developers Conference). 2012. URL: https://www.gdcvault.com/play/1015676/Terrain-in-Battlefield-3-A.

[23] Jian Wu et al. "A New Quadtree-based Terrain LOD Algorithm". In: (2010). URL: http://www.jsoftware.us/vol5/jsw0507-12.pdf.