Technische Hochschule Ingolstadt
Faculty for Computer Science
Field of Study: Computer Science
MBDA Deutschland GmbH

Vulkan Terrain Renderer Based on Real-World Height-Maps for Embedded Systems

Bachelor's Thesis

Peter Fischer
Matrikelnr.: 00115030
E-Mail: peter.fischer.w@gmail.com

1st Examiner	Prof. Dr.-Ing. Richard Membarth
2nd Examiner	Prof. Dr. Torsten Schön
Supervisor	Thomas Britzelmeier
Issued on	30.01.2024
Submitted on	01.03.2024

Declaration

I hereby declare that this thesis is my own work, that I have not presented it elsewhere for examination purposes and that I have not used any sources or aids other than those stated. I have marked verbatim and indirect quotations as such.

Ingolstadt, March 1, 2024

Peter Fischer

Abstract

Real-time rendering of real world terrain states a popular field of research of the recent decades. Real world terrain is represented by highly detailed data sets. Therefore, it is crucial to use methods and algorithms that create simplified terrain meshes out of this data. As a result, steadily new approaches and techniques are developed, to render terrain efficiently with a high degree of detail, while trying to use the most of the current hardware's performance.

The goal of this thesis is to develop an off-screen terrain renderer using Vulkan as graphics API and real world data sets, as height maps and satellite images. Additionally, the developed terrain render is supposed to be executed on selected embedded devices, while still delivering high framerates.

In order to achieve this goal, necessary basic terminology and concepts of terrain rendering are dealt with. An important part of these concepts are performance optimizations, as level of detail and culling methods.

Furthermore, Vulkan as graphics API will be introduced and compared with its predecessor OpenGL. For a technical understanding of Vulkan it is crucial to look at its render pipeline and basic rendering concepts.

After several rendering methods are analyzed, based on them the design of this thesis's renderer is examined, while the most important parts of its implementation are looked at closer by analyzing their code snippets.

The main part of this thesis is completed, by analyzing benchmarks of different scenarios to see how the developed terrain render behaves in different situations. These benchmarks include the frames per second, and statistics of the device's hardware components, as CPU load, GPU load, memory usage and memory throughput.

The thesis ends, with a summary and general conclusion of the work. Possible future work includes optimizing the usage of embedded devices' unified memory, testing of texture compression methods for an optimized memory usage or a comparison of an OpenGL version of the developed renderer by comparing benchmark results of the same scenarios.

Contents

1 Introduction

1.1 Motivation

Terrain rendering is a popular field of research that researchers have been conducting research in recent decades. Already in 1997, one of the first prominent terrain rendering methods were developed [14]. With the rapidly increasing development of technology, especially consumer graphics cards, it was important to adapt to the new circumstances in order to use new hardware efficiently. Therefore, rendering methods shifted from CPU heavily implementations to techniques that avoid high CPU overhead and take advantage of more powerful GPUs. The challenge here is to present the physical reality with large data sets, consisting of height maps and satellite images, efficiently, but still with a high degree of detail. To solve this challenge, several different approaches have been developed on how the terrain's geometry should be created from extracted data of height maps. When it comes to embedded devices, there is another challenge to achieve optimal results with limited hardware resources. Terrain rendering plays an important role for video games, visualizations and simulations. The latter point is probably the most important point in relation to this thesis and for the usage of MBDA. With real-time rendering of the real world, 2D maps of previous simulations could be replaced by more appealing 3D representations.

1.2 Goal and Structure of this Thesis

The goal of this bachelor thesis is to develop an off-screen terrain renderer that uses the graphics API Vulkan. The rendered terrain should represent the real world, by using real world height information from height maps and satellite images. Furthermore, it should be capable of running on embedded devices, while delivering a decent amount frames per second.

The thesis first establishs a technical background, by describing and explaining basic concepts, that are necessary for the development a terrain renderer.

Chapter 3 introduces the graphics API Vulkan, along with its render pipeline and basic concepts that are necessary for rendering images.

Afterwards, in chapter 4, several terrain rendering methods are described, that were developed over the course of time, and serve as a basis for the design decision of the developed terrain renderer of this thesis.

In chapter 5 and 6, the methods of the terrain renderer are explained and core parts of the implementation are shown and discussed, by analyzing their code snippets.

For analyzing the developed terrain render regarding its performance and the behavior of the hardware components, benchmark results of different scenarios are used in chapter 7.

The thesis is completed, by a summary and an outlook to future works in chapter 8.

2 Technical Background

Before being able to analyze and introduce existing rendering methods a technical background has to be established. At first, basic terminology of graphics programming will be explained. Next, triangulation and different level of detail concepts are introduced, which state a significant importance when it comes to the approximation of terrain meshes. Furthermore, different culling methods are explained that represent a part of rendering performance optimizations. Lastly, real-world data that is commonly used for terrain rendering is presented.

2.1 Graphics Programming Terminology

In this section, the definition of a few basic graphics programming terminologies is given. Further explanations will base on these terms. Therefore, it is important to know them.

2.1.1 Vertex

A vertex is a data structure used in computer graphics. It can contain multiple attributes, while the most important are coordinates of the vertex position, color values and texture coordinates, that are used for texture mapping on surfaces. Vertices are the most fundamental part in computer graphics and are used for building primitives, like triangles or quads, which are linked together to form meshes. [20]

2.1.2 Vertex and Index Buffer

A vertex buffer holds data of multiple vertices, like position, color and texture coordinates, which can be used for rendering. A buffer like this is usually located in device-local memory instead of host-visible memory. Device-local memory represents a part of the graphic card's memory, also called VRAM, which can not be accessed directly by the CPU, while host-visible memory is part of the VRAM, which is visible and can therefore accessed by the CPU. The difference between both is, that device-local memory delivers higher performance when it comes to rendering processes.[24]
When rendering 3D meshes, which consists of triangles, vertices are shared between multiple triangles. In case of using a vertex buffer, it has to contain multiple data of the same vertex, which leads to redundancy and unnecessary memory usage. To avoid these circumstances, the usage of an index buffer offers a good solution. An index buffer stores an array of pointers into a corresponding vertex buffer. This allows to use the same vertex data multiple times, without the need of having multiple copies of that data. [38] Figure 2.1 demonstrates the drawing of a rectangle with only a vertex buffer and vertex buffer in combination with an index buffer.

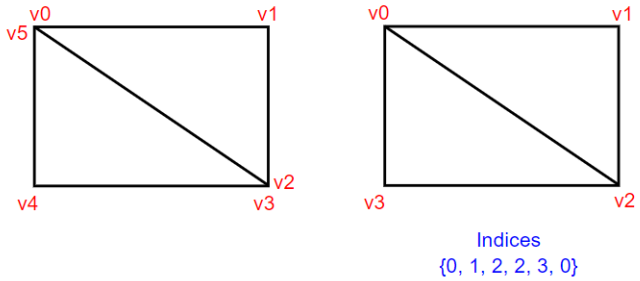


Figure 2.1: Drawing of a rectangle with vertex buffer (left) and vertex buffer with index buffer (right). (taken from [38])

2.2 Coordinate Systems

Vertex coordinates of an object or mesh need to undergo a step-by-step transformation between different coordinate systems, also called spaces, in the rendering process. The transformations between the coordinate systems are performed with transformation matrices. The overall goal is to combine multiple objects or meshes in a mutual coordinate systems. The final coordinates should be normalized coordinates in the range of [-1.0, 1.0], also called normalized device coordinates (NDC). These are used in the final step to transform them to 2D pixels on the screen. The reason for using different coordinate systems is that some operations are easier to perform in certain spaces. Therefore every coordinate system in this process has a specific use and phase where it is being used. [53] Figure 2.2 gives a quick overview of the transformation sequence.

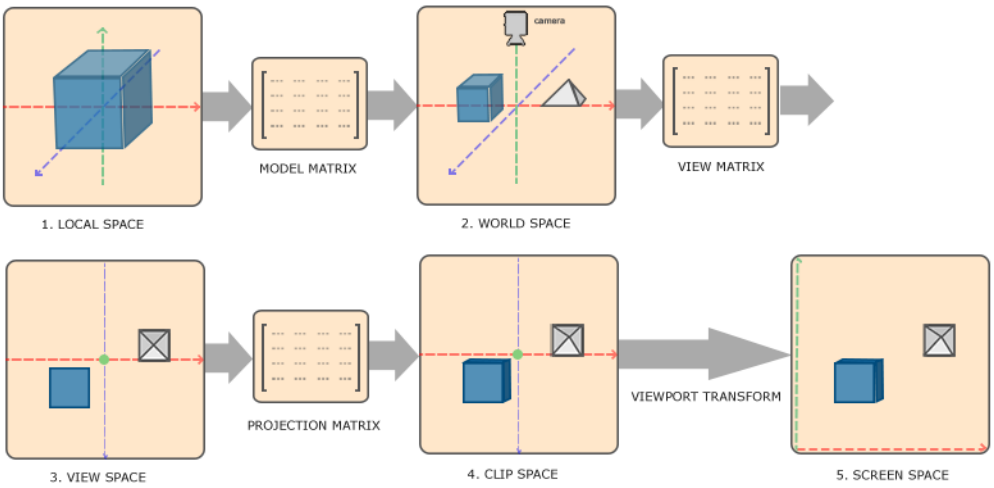


Figure 2.2: Sequence of coordinate system transformations.(taken from [53])

2.2.1 Model space

The model space, also called local space, contains local coordinates, which are coordinates of an object relative to its own local origin. Each object has its own model space, which makes it easier to apply transformations, like rotation, scaling or translation, on an object without influencing other objects in a scene. [53]

2.2.2 World space

The world space represents a coordinate system of the actual world of the scene. It can contain multiple different objects, while each object of a scene is placed relative to the origin of the world space. This space is used to define each object's location and orientation in a scene.[53]

2.2.3 View space

The view space reflects the camera's point of view. World space coordinates are transformed to view space coordinates in order to define how the scene should be rendered from the view of the camera. Hence the coordinates are no longer relative to the world's origin, but relative to the center of the field of view.[53]

2.2.4 Clip space

In clip space all coordinates are transformed and prepared for clipping, a process that can determine whether a vertex is inside the field of view or not. All vertices that are outside of the field of view are not visible, get discarded and are not considered for further actions. This process gives the clip space its name. After clipping is performed, coordinates are converted to NDCs for the next phase.[53]

2.2.5 Screen space

After the vertices of a scene are transformed to NDCs, the transformation to screen coordinates takes place. In this process the size of the screen is taken into account to represent the scene correctly. The result of the transformation are 2D coordinates that correspond to a pixel of the screen.[53]

2.3 Triangulation

Triangulation of terrain mesh models is the core part of rendering terrain. When it comes to graphics APIs, triangles are the basic concept of representing models or surfaces with them being arranged in meshes. Therefore, methods for creating meshes that represent the surface of the terrain and consist of vertices which are linked to form triangles are necessary. The information for the vertices are extracted from height maps like digital elevation or surface models which will be explained later on in Section 2.6. The height map's values define the terrain's shape. A first approach would be using every point of the map as a vertex and creating a triangle mesh from it. Unfortunately this simple approach is impractical. When having a height map with a resolution of one meter the amount of triangles being rendered is becoming too large, which leads to performance issues. Furthermore, there is no need for terrain which is far away to have the same amount of triangulation as terrain which is close to the camera. Additionally, areas which have barely elevation differences do not need to be triangulated with a high resolution. In these cases the number of triangles which represent these parts of the terrain can be decreased, which leads to a gain of performance. Figure 2.3 illustrates the mentioned issues with this approach. [17]

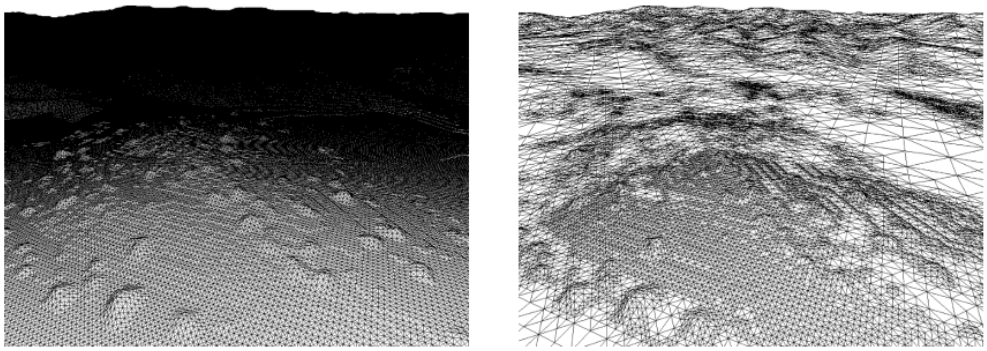


Figure 2.3: Difference between initial geometry from elevation map and simplified geometry. (taken from [26])

As a result methods are necessary to create simplified meshes which contain as few triangles as possible while still representing the terrain accurately. Triangulated irregular networks (TINs), right-triangulated irregular networks (RTINs) and regular grids are the most common approaches for creating such meshes. [26], [17]

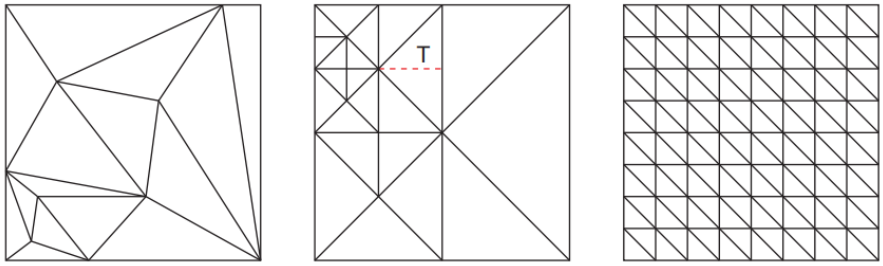


Figure 2.4: Comparison of triangulation approaches. (taken from [26])

2.3.1 Triangulated Irregular Networks

TINs do not follow any constraints and therefore building completely irregular meshes. The mesh's vertices are represented by their three coordinates and since TINs do not have any constraints, it's the model which contains the lowest number of triangles, while still representing the terrain accurately. [26] The main idea behind the triangulation of a TIN is to carefully select which samples of the height map's rastergrid need to be used. Areas with major changes in their shape need to use a larger amount of height samples than flat areas. One downside of using TINs is that they are costly in computation. When creating a TIN every sample of the raster grid needs to be considered, which leads to very large numbers of CPU computations. [3]

2.3.2 Right-Triangulated Irregular Networks

RTIN is a widely used mesh for representing terrain. Compared to TINs the vertices are positioned on the same regular grid as the sample values from the original height map. Another characteristic is that all triangles are right isosceles triangles when looking from the top, which allows the mesh's update process, which consists of splitting and merging the triangles, to be easier and much faster. However this approach needs a larger amount of triangles to represent the same terrain than with TINs, which results in a higher consumption of memory usage. [26] The basic idea behind the creation of RTINs is the subdivision of triangles based on longest edge bisection. This means that triangles get split along their hypothenuse, which results in having two smaller right isosceles triangles. The procedure starts with a simple mesh of two or four right isosceles triangles which cover the area of the terrain, on which an adaptive recursive refinement of the mesh is performed. Certain refinement criteria specify whether a triangle gets split or not. One refinement criteria could be whether a triangle approximates the corresponding part of the original mesh well enough. [27] An example of how the splitting of triangles works is illustrated in Figure 2.5.

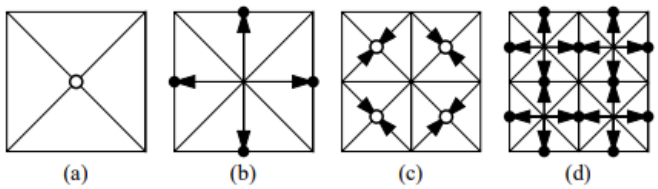


Figure 2.5: Demonstration of triangle splitting. (taken from [27])

2.3.3 Regular Grids

The usage of regular grids is one of the simplest approaches when it comes to triangulation of terrain. With this approach a vertex gets placed on every point of a regular 2D grid. Afterwards, each vertex gets displaced in its height by the corresponding sample from the original height map. Since it is a regular grid, triangles can be put together easily and the number of triangles depends on the size of the grid. A negative aspect of this approach is, that the triangulated mesh does not approximate the original model as good as the other two approaches. Nevertheless, this method can create a simplified version of the original mesh with small computational effort. [26]

2.4 Level of Detail

Level of Detail (LOD) algorithms are tightly connected with the triangulation and can be regarded as a performance optimization. They are used to determine the complexity of a mesh's geometry parts and therefore reducing the amount of triangles that are used in a mesh. The decision of a LOD which should be used for a mesh geometry is also referred to LOD management. LOD management can be succeeded by different error metrics. A simple way to determine the LOD is based on the distance between the viewpoint and the part of the mesh. Another method is called screen-space error. The screen-space error measures the difference in pixels between the actual representation of an object on the screen and its simplified version. [1]

2.4.1 Discrete Level of Detail

An approach for a LOD algorithm is discrete level of detail (DLOD). The DLOD method holds multiple versions of a mesh in different levels of detail. These simplifications of a mesh are pre-processed in an offline computation. During run-time an algorithm decides which simplified mesh is most suitable for the current situation. Since for distant objects a less detailed version is used, the number of polygons can get reduced to gain a higher rendering speed. A DLOD algorithm is also considered as view-independent LOD, because it cannot check from which direction an object is looked at. Therefore the simplification of a LOD is applied equally over the whole object's mesh. [1] A downside of using DLOD is the vulnerability of popping effects. Popping effects are unexpected visual effects that occur between abrupt transitions of different LOD. [21]

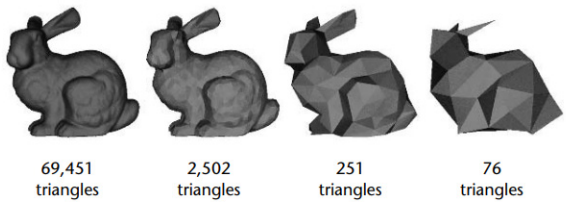


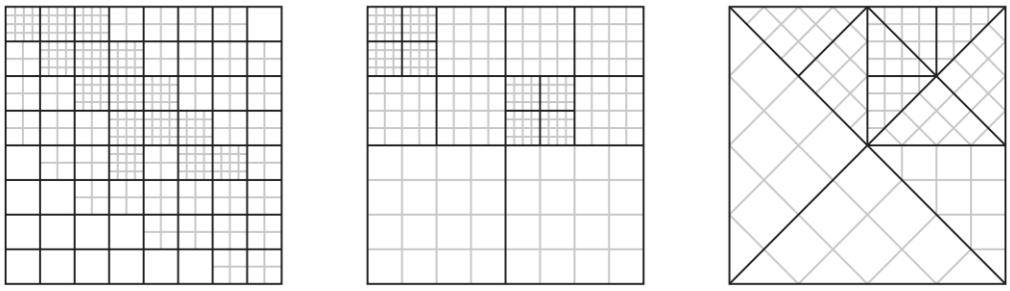
Figure 2.6: Mesh of a bunny in different discrete levels of detail. (taken from [1])

2.4.2 Continuous Level of Detail

Continuous level of detail (CLOD) is another approach for LOD. Compared to DLOD, with this method no pre-computed meshes are used for different simplification levels. Instead an algorithm specific data structure holds an encoded spectrum of detail levels. The system then decides during run-time what data is most suitable and extracts the desired data from the data structure. [28] The LOD can be defined in a more precisely way for each object, what makes CLOD superior to DLOD in terms of better granularity and smoother transitions between the switching of LODs. Furthermore, an increase of performance can be expected, due to a more precisely LOD selection, what secures that only as many polygons as needed are used for rendering a mesh. [1]

2.4.3 Multi-resolutional Level of Detail

Multi-resolutional level of detail (MLOD) is an approach of rendering terrain based on defined blocks that are put together. MLOD uses a structure of a block hierarchy, where each block is recursively split into a greater amount of smaller sized blocks. These blocks can be either in a rectangular or triangle shape. In case of rectangular blocks a quadtree is used to to represent the block hierarchy, while for triangular blocks a binary tree is used. Each block relates to a pre-computed LOD. During run-time an algorithm decides based on screen-space error what block size is most suitable for representing an area of terrain and loads necessary blocks from the structure. A characteristic of this approach is that every block independent of its size contains the same amount of triangles. Therefore when splitting a block in its child nodes the represented terrain area is rendered with a larger amount of triangles and thus in a higher detail. In case of a quadtree structure this leads to a four times higher resolution of an area, when splitting a block, while using triangular blocks in a binary tree ends up in doubling the resolution. [26]



(a) Fixed-size blocks (b) Quad-tree (c) Binary tree

Figure 2.7: Comparison of different MLOD approaches. Black lines represent a block and grey grids its geometry. (taken from [26])

2.5 Culling Methods

When rendering a scene not every object or part of objects are visible in the Field of View (FOV). Rendering of these objects causes a loss of performance due to not being visible on the screen or rendered image. Therefore methods for visibility culling will be introduced. The goal of these methods is to reduce the amount of vertices being rendered and saving compute power and gain performance as a result. [51]

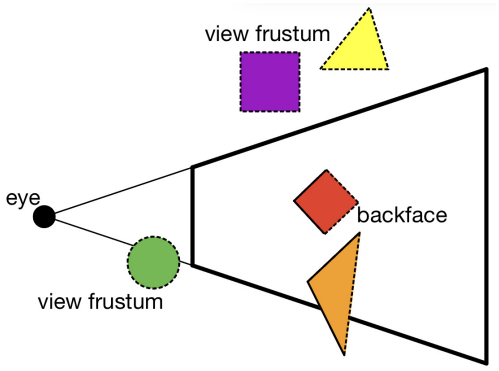


Figure 2.8: Demonstration of culling methods. Dotted lines represent culled surfaces. (adapted from [35]

2.5.1 Frustum Culling

Frustum culling is an object based culling algorithm. The idea behind the algorithm is while rendering an image to discard every object located outside the current viewing volume whose geometry is a frustum. The frustum consists of six clipping planes. If an object is within the frustum it is marked as visible and gets rendered, else the object will be marked as invisible and gets culled. Whenever an object is only partly within the frustum, it is marked as visible as well. [56]

2.5.2 Back-Face Culling

Back-Face culling is a method with the goal to decrease the amount of rendered triangles. For triangles which are facing away from the camera is no need for being rendered. With the help of backface culling these triangles can be detected and avoided to get rendered. There are two common approaches to find triangles facing away from the camera. The first one uses the surface normals of a triangle. For determining whether a triangle is back facing or not the computation of the dot product of the triangle's normal with a vector from the view-point to any point on the triangle can be used. If the result is greater than zero, the triangle is determined as back facing and gets culled. [23]

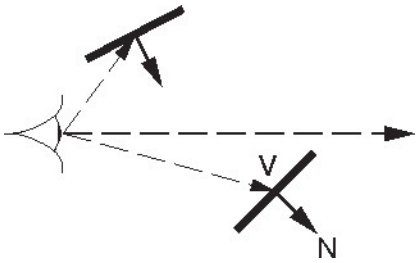


Figure 2.9: Demonstration of backface culling with the computation of the dot product approach. V is a vector from eye to point on triangle. N is the triangle's normal. (taken from[54])

The second approach uses the winding order of triangles. A triangle can have a clockwise or counter-clockwise winding, depending on the given order of the triangle's vertices. A triangle with a clockwise winding means that the three vertices rotate clockwise around the triangle's center and vice versa. The order of the triangle's vertices are determined in an early stage of the graphics pipeline, called input assembler, which will be explained together with the other parts of the rendering pipeline in section 3.2. In a default OpenGL and Vulkan context the front faces are counter-clockwise, but may change depending on the implementation. In order to perform backface culling with the second approach it is just necessary to cut out all triangles which winding do not correspond to the front face's winding. This approach of backface culling is implemented in the rendering pipeline of OpenGL and Vulkan. [15]

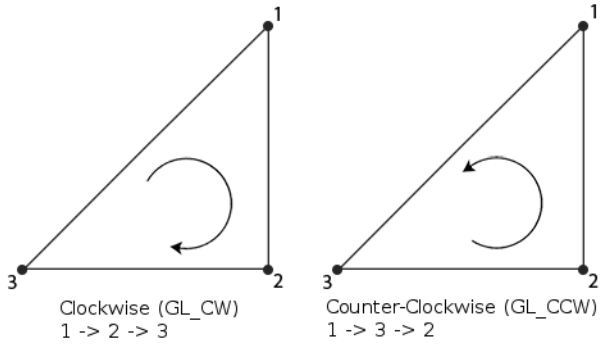


Figure 2.10: Demonstration of winding orders. (taken from [15])

2.6 Real-World Data

The following section will deal with different height models and textures that are used later on as data for the real world terrain.

2.6.1 Elevation Model

Height information is an important part when it comes to rendering terrain and can be represented in different ways. One of them is the digital elevation model (DEM), also called digital terrain model. This model represents the bare ground topographic surface of the earth with its elevation data organized in a regular raster grid. Each sample of the raster grid has a height value that is relative to sea level. DEMs do not include any information about buildings or vegetation. Another common model for representing elevation data is the digital surface model (DSM). DSMs are similar to DEMs. While DEMs only hold information about the earth's bare ground surface, DSMs height data includes information about structures and vegetation, like houses or trees. Both models can be in different formats, like GeoTiff, greyscaled heightmaps or USGS DEM. Figure 2.11 illustrates the difference between both models. [19]

2.6.2 Digital Orthophoto

For visualizing terrain, images of the earth's surface are necessary. Digital orthophotos (DOPs) are suitable resources to use for mapping textures on terrain models. DOPs are fused aerial and satellite images, which are geometrically corrected. Distortions caused by terrain surfaces and camera lenses are removed and angles from cameras' position are compensated. The idead behind DOPs is the creation of surface images where distances are uniformed across the image. Figure 2.11 shows an example of a DOP. [33]

Figure 2.11: Difference between DEM (blue) and DSM (red) on the left (taken from [13]) and an example of a digital orthophoto (taken from [50]) on the right.

3 Vulkan

Vulkan is a new-generation open-source graphics and compute API which was first released in 2016. With the release of Vulkan, developers now have next to OpenGL a open-source graphics API with similar target platforms from which they can choose. [10] However, Vulkan is the first new-generation graphics API which enables cross platform portability. Therefore Vulkan applications can run on mobile devices, without the need of a different implementation. [11] OpenGL on the other hand has OpenGL ES as a subset for implementing applications for mobile devices or other devices with simpler and less powerful hardware. [34] The idea behind Vulkan was to release a successor for OpenGL which implements modern techniques and concepts resulting in better performance. Nevertheless, since OpenGL still has a large user group, companies like NVIDIA will continue to maintain and support OpenGL and it will coexist next to Vulkan for a while. [11]

3.1 Vulkan vs OpenGL

Since Vulkan is touted as OpenGL's successor it's helpful to have a look at their differences to understand why Vulkan has better performance in most cases.
The most significant difference is Vulkan's simpler and thinner driver, which grants less latency and overhead. This is achieved by the reduction of automatic memory and error management. While error management is always active in OpenGL, in Vulkan developers have to enable it in the first place to use it, therefore being able to run the application in a release mode without error management. The reduction of the driver comes with a greater responsibility for the developer especially in terms of memory and thread management. Developers have to provide way more low-level information to the application when using Vulkan, which is also incidental to its reduced driver. Furthermore Vulkan grants a better utilization of the GPU. This is due to efficent multithreading capabilities with multi-core CPUs. Also compared to OpenGL the driver has to perform less CPU work before passing on work to the GPU. Another befenefit of using Vulkan is the capability of off-screen rendering, which will be used in this thesis's renderer. Vulkan is designed for being able to perform rendering work without the necessity of a window surface, while OpenGL requires to have a window object in its context. However, this does not mean, that off-screen rendering is not possible with OpenGL, but workarounds are necessary for OpenGL being able to render off-screen. [10] Vulkan comes with a few downsides compared to OpenGL as well. Developing a Vulkan application will appear more complex. In order to see any visible results on the screen, a far longer developing process is necessary, which is a result of Vulkan's thinner driver. [2] In addition some cases have shown, that embedded hardware may cause incompatibility with certain Vulkan extensions. As a reason, it could be assumed that Vulkan is a newer graphics API which extensions and functionalities are yet, in spite of a platform independent design, not fully supported by all platforms.

3.2 Render Pipeline

The render pipeline is the core part of every graphics API, with the same basic design. It represents a sequence of operations that are necessary to render images. The pipeline takes data like vertices and textures as an input, which are then processed in multiple stages, and finally returns pixels of an image. The render pipeline can be considered as an assembly line, where every stage performs an operation on the data and passes it on to the next stage. The pipeline consists of fixed-function stages and programmable stages. Fixed-function stages' behavior can not be changed, but parameters can be used to adjust the operations in these stages. On the other hand programmable stages are specified what they should do, but they can be fully modified by using OpenGL shading language code to apply exact behavior to these stages. [37]

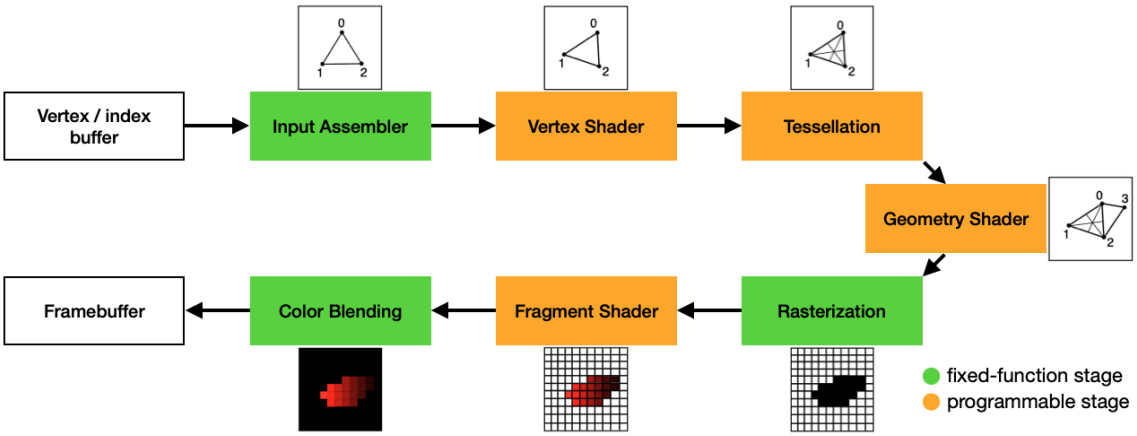


Figure 3.1: Render Pipeline (adapted from [37])

3.2.1 Input Assembler

The input assembler is the first stage of the rendering pipeline and is responsible for the preparation and organisation of the supplied vertices. Its purpose is to read the raw vertex data from a specified buffer. Additionaly it is possible to add an index buffer to avoid the repetition of vertices. Based on the given primitive type parameter the input assembler assembles all collected vertices to primitives, which are the base for all of the upcomming stages. Examples for primitive types are lines, points or triangles, while triangles are mainly used in graphics programming. [43]

3.2.2 Vertex Shader

The vertex shader's task is to apply transformations on each vertex to transform its position from model space to screen space and is a programmable stage. These transformations are performed on each vertex indiviually and each input vertex, which is received from the input assembler's ouput, must map to a specific output vertex. [43]

3.2.3 Tessellation

Tessellation is a optional pipeline stage and consists of two programmable shaders and one fixed-function stage in between. The main idea behind tessellation is to subdivide patches of vertices into smaller primitives and compute new vertex values (position, color, texture coordinates, etc.) for each generated vertex. This process can be used to increase an object's geometric details and is useful when it comes to illustrating complex surfaces. Patches, on which tessellation performs subdividing, are a primitive type, where *n* vertices represent a patch. [47]

3.2.3.1 Tessellation Control Shader

The Tessellation Control Shader (TCS) is the first step of tessellation. The TCS receives patches of vertices and is responsible for defining how precise each patch should be subdivided and applies transformations on the input patches if necessary. The usage of a custom TCS for tessellation is optional. If no TCS is specified by the developer, the patch data gets automatically passed on from the vertex shader to the tessellation primitive generator. Default values then decide how precise tessellation should be applied to each patch.
One of the TCS's output are the tessellation levels consisting of a two vectors which define the outer and inner tessellation. The size of the vector depends on the selected primitive type and describes the amount of levels. When using a triangle as primitive type, the inner tessellation vector uses one level and the outer tessellation vector uses three levels. The basic idea behind tessellation levels is to define of how many segments a edge should get tessellated to. A tessellation level of four causes an edge getting tessellated to four edges. Figure 3.2 illustrates this process. However the tessellation levels in the way they are specified by the TCS are not directly used in the next stage. The defined tessellation levels from the TCS will be modified by a few other factors to get the final tessellation levels which are used to tessellate the primitives. This process and its depending factors will be explained in detail in the section of Tessellation Primitive Generation.[48]

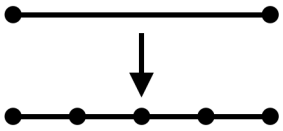


Figure 3.2: Illustration of tessellation level. (taken from [48])

3.2.3.2 Tessellation Primitive Generation

Primitive generation is a fixed-function stage which sits in between the tessellation control shader and tessellation evaluation shader. Since the TCS is a non mandatory stage for tessellation, the primitve generation only gets executed when an active tessellation evaluation shader is used in the application. The primitive generation's task is to create a set of new primitives for each input patch. The way how primitve generation works is affected by several factors. The first one is tessellation levels, which can either be an ouput from the TCS or set by default values. The other factors depend on the tessellation evaluation shader's input. The relevant input types are the spacing of the tessellated vertices, the input primitive type and the primitive generation order. [48]
The output of the primitive generation stage are no vertices with vertex data like position or color, as

you might expect. This step of the tessellation stage does not refer to the actual vertex coordinates of the patches which are supplied by the TCS or the vertex shader if no TCS is being used. The only factors which are taken into account are the above mentioned factors tessellation levels, tessellated spacing, primitive type and order. The output of the primitive generation are generated vertices with a normalized position in range of [0, 1]. The generated normalized vertices are supplied in the built-in *gl_TessCoord* vector, which will be further processed by the tessellation evaluation shader. As earlier mentioned tesselation levels determine how many new vertices are created, but the tessellation levels received from the TCS still have to be modified. This process depends on the spacing parameter defined in the tessellation evaluation shader. The spacing parameter can be *equal_spacing*, *fractional_even_spacing* or *fractional_odd_spacing*. Depending on which spacing parameter is defined, the visual results differ. While the results with *equal_spacing* appear to have popping effects as tessellation levels change, *fractional_even_spacing* and *fractional_odd_spacing* results in a smoother behavior. In case of *equal_spacing* each tessellation level is rounded up to to the nearest interger, which represents the effective tessellation level and all tessellated segments from an edge have the same size, which is equal to Figure 3.2. When using *fractional_even_spacing* each tessellation value is rounded up to the nearest even integer, while in case of *fractional_odd_spacing* the value is rounded up to the nearest odd integer. In both cases an edge consists of two sets of segments. The first set contains *n-2* Segments, where *n* is the effective tessellation level, while the other set holds two segments. Segments in the same set have always the same length, while segments in the second set are shorter. The length of the the second set's segments is relative to others, on base of *n-f*, where *n* is again the effective tessellation level and *f* represents the tessellation level calculated by the TCS. Figure 3.3 gives an basic idea of how both methods work. [48]

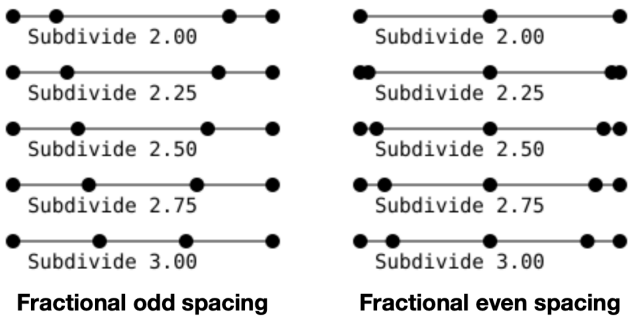


Figure 3.3: Illustrates fractional_odd_spacing and fractional_even_spacing. (taken from [48])

As already mentioned, when it comes to tessellation of a triangle as abstract patch, three outer and one inner tessellation is used. Applying outer tessellation levels is a straight forward process. Each level is applied to one edge of the triangle. However, applying the inner tessellation level needs a bit more complex process. The algorithm for the inner tessellation ignores all outer tessellation levels, and subdivides all three edges based on the inner level. At every corner the two newly created neighboring vertices are taken and used to compute a new vertex, based on the intersection of their perpendicular lines. Based on these vertices a new inner triangle is created. Perpendicular lines from all other vertices of the outer edges are used to determine, where each edge of the inner triangle gets subdivided. This process is repeated with the newly created triangle until either the triangle consists only of three vertices or each edge has two subdivided edges. In the former case the process stops, while in the latter a single vertex in the center of the most inner triangle is generated and stops as well. Figure 3.4 visualizes this described process. [48]

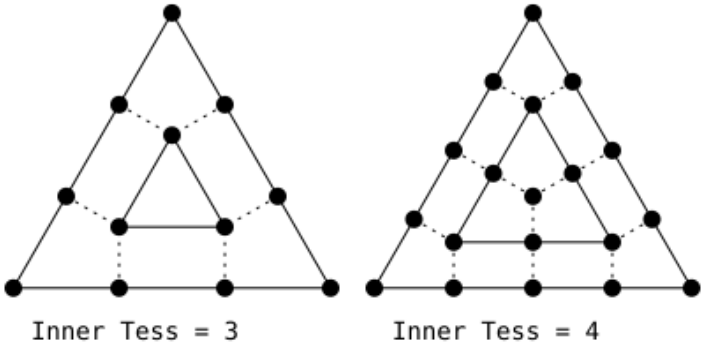


Figure 3.4: Applying of inner tessellation level on triangle. (taken from [48])

After inner tessellation is done, the subdivision on the outer triangle is removed and re-tessellated with the corresponding outer tessellation levels. All generated vertices are then needed to connect to triangles based on specified criteria, as that the whole area of the patch is covered with triangles and no triangles overlap each other. [48]

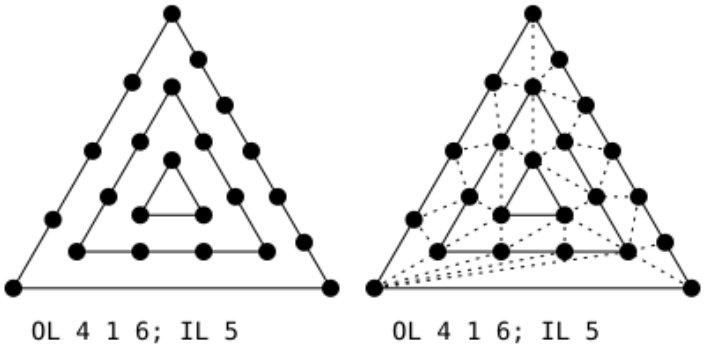


Figure 3.5: Tessellated triangle with vertices connected to triangles (right). (taken from [48])

3.2.3.3 Tessellation Evaluation Shader

The Tessellation evaluation shader (TES) is the final part of the tessellation stage and is a programmable stage, which is, not like the TCS, a mandatory stage for tessellation. In here are necessary parameters (prmitive types, tessellation spacing and order) defined which are required in the primitive generation stage. The TES receives generated vertices with normalized position data from the primitive generation and the output values of the TCS. With these data the TES computes all necessary values for each vertex particularly the actual position in the clip space. [48]

3.2.4 Geometry Shader

Geometry shaders is another optional programmable stage, which is capable of processing and manipulating geometries of primitives. The geometry shader is executed for every primitive and it is able to discard or add more primitives to its output. [18] It can be used for similiar operations as tessellation does, however due to its bad performance on most graphic cards it is not recommended to use that shader in applications. [37]

3.2.5 Rasterization

Rasterization is a fixed-function stage, which is responsible for transforming 3D geometries to a 2D image. The rasterization stage receives primitives which consist of clip-coordinates and checks which square of a grid in framebuffer coordinates is occupied by each primitive. Based on depth values which are assinged to every square, the rasterization stage can later on determine which primitive gets assigned to the square. Each of these squares represents a fragment, which is used to compute the final data for a pixel. [43] The rasterization stage also performs methods like back-face culling and clipping to drop all primitives which cannot be seen by the camera. [42]

3.2.6 Fragment Shader

The fragment shader is another programmable stage in the rendering pipeline. In this stage every fragment which passes the rasterization stage is processed. Each of the received fragments correspond to a potential pixel of the screen. The fragment shader is invoked for every fragment individually and its main target is to compute final color and depth values for the processed fragment. It has access to texture data, which grants the shader to color fragments based on texture images and received texture coordinates. The output consists of a color value and depth values, while latter is used for depth testing. [16]

3.2.7 Color Blending

The color blending stage is a fixed-function stage and also the final stage of the rendering pipeline, before finally writing values to the framebuffer. It takes the fragment colors from the fragment shader's output. In this stage multiple processes take place. This can either be the mixing of fragments which map to the same pixel in the framebuffer and blending operations, between the source color, which is written to the fragment by the fragment shader, and a destination color, which is a color from the framebuffer. When creating the pipeline in the application, the exact behavior of the color blending's processes can be defined by setting parameters for this stage. After a fragment is processed by the color blending stage, the final color value will be written to a color buffer, which is part of the framebuffer. [36]

3.3 OpenGL Shading Language

OpenGL shading language (GLSL) is a C-style language which is used for programming programmable stages in the graphics pipeline. Since it is used for graphics programming it also includes and supports built-in vector and matrix primitives and functions for calculation operations with vectors and matrices. Although GLSL was initially made for OpenGL, it is also used for shader programming in Vulkan. [8]

3.4 Introduction of Basic Vulkan Concepts

In the following section a basic introduction on fundamental objects and concepts are given, that are necessary to render images in Vulkan. This deals with the creation of a simple image to concepts, on how to provide resources and data to shader stages in the rendering pipeline. The goal is to establish a basic understanding on how these things are done when using Vulkan, so that the explanation of the used rendering method and parts of the implementation in later chapters are comprehensible.

3.4.1 Images and Samplers

Creating an image in Vulkan consists of multiple steps and needs three different objects, as VkImage, VkDeviceMemory and VkImageView. A VkImage holds the actual data of the image and needs to be bound with a VkDeviceMemory object. A VkDeviceMemory object is used for allocating the necessary memory for a VkImage. VkImage objects can not be accessed directly by for example shaders, because the object does not give any information on how the image can be accessed. [6] Therefore, a reference to an image is necessary, that supplies more information on how a specific image can be used or accessed. For those needs a VkImageView object is used, which defines how the data of the corresponding image can be interpreted. [52]

Additionally, for the usage of images in shaders a VkSampler object is necessary. The official Vulkan documentation describes a sampler as an object that is used to read image data in shaders and is used in combination with a VkImageView object. [45]

3.4.2 Framebuffer

A framebuffer is stored in Vulkan in a VkFramebuffer object. It consists of multiple memory attachments, which usually are image views. Framebuffers are bound to the graphics pipeline and define among other things, which can be ignored by now, a target image. This target image is used as a destination for the rendered images. [32]

3.4.3 Uniform Buffer

A uniform buffer is a buffer used by shaders. The idea behind this type of buffer is to supply shaders a set of parameter data. Common parameters that are stored in uniform buffers are the model, view and projection transformation matrices. [31]

3.4.4 Descriptor Sets

So far, it was explained what objects, buffers or data can be provided to shaders. However, there is the detail to clear up on how shaders can access all of these things. In order to supply data to shaders, Vulkan specifies the usage of descriptor sets. As the name suggests, a descriptor set holds multiple descriptors, where a descriptor is a data structure representing a resource, that should be provided to shaders. These resources can be in form of a buffer, an image view or a sampler. A descriptor set can be bound during a command recording phase, to use the descriptors' resources while rendering. For being able to create a descriptor set, that is stored in a *VkDescriptorSet* object, a descriptor set layout has to be established first, which is stored in a *VkDescriptorSetLayout* object. This object describes how a descriptor set is structured and has binding indices, which will be used by the shaders to access the required resource. [44] Code 3.1 shows an example on how to create a descriptor set layout with an image sampler as resource.

```
1  VkDescriptorSetLayoutBinding textureSamplerLayoutBinding{};
2  // specifies binding for shader access
3  textureSamplerLayoutBinding.binding = 0;
4  // specifies the type of the resource
5  textureSamplerLayoutBinding.descriptorType =
       VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
6  textureSamplerLayoutBinding.descriptorCount = 1;
7  // specifies in what shader stage descriptor can be used
8  textureSamplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
9
10 array<VkDescriptorSetLayoutBinding, 1> bindings = {textureSamplerLayoutBinding
       };
11
12 VkDescriptorSetLayoutCreateInfo layoutInfo{};
13 layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
14 layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
15 layoutInfo.pBindings = bindings.data();
16
17 // creates descriptor set layout and stores it to a given reference
18 vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout)
       ;
```

Code 3.1: Example of a descripor set layout.

Before a descriptor set of a given layout can be created, a descriptor pool must first be created. A descriptor pool is necessary for allocating descriptor sets. [4] It defines how many descriptors of each type and descriptor sets in general can be allocated from its pool. [44] Figure 3.6 visualizes the allocation of descriptor sets, by considering the three mentioned parts.

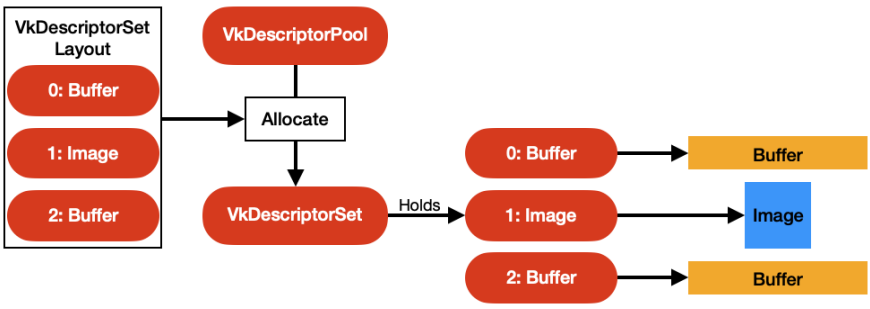


Figure 3.6: Shows process of creating a descriptor set. (adapted from [4])

3.4.5 Queues

Queues can be described as an interface between the application and the GPU. Whenever an application submits work to the GPU, it does this by assigning the work to a queue. The GPU can then access the loaded work from the queue and starts processing it. Vulkan specifies four different types of queues, as a graphics, compute, transfer and sparse queue, where each is used for different kind of tasks. Furthermore, queues are bundled in queue families, where each family supports at least one of the mentioned queues. Therefore, before submitting work to a queue, the GPU's queue families have to be queried and checked what queue family contains the necessary queue types for the work. [46]

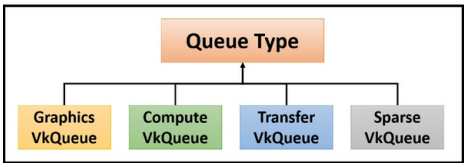


Figure 3.7: Shows Vulkan's queue types. (taken from [46])

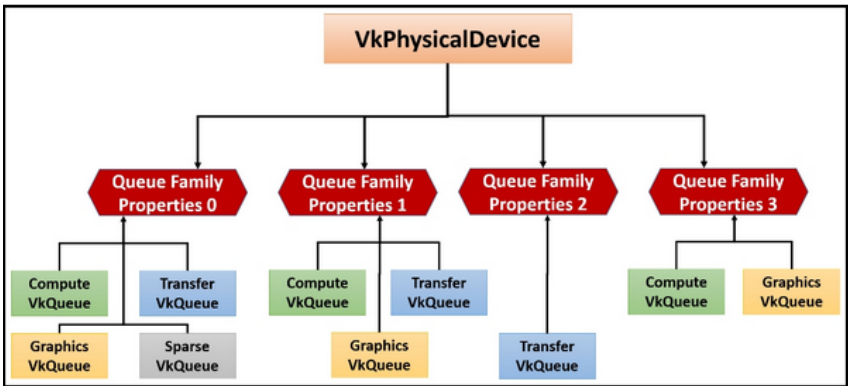


Figure 3.8: Shows an example of possible queue families. (taken from [46])

3.4.6 Command Buffer

Command buffers represent objects for recording commands and submitting them for execution. They are used for multiple tasks, as changing an image's layout, copying data from a buffer to an image or submitting work to the graphics pipeline for rendering frames. Creating, recording and submitting a command buffer is represented by a general sequence of steps.

At first, a command buffer needs to be allocated from a command pool. A command pool is an object that is used for allocating the necessary memory a command buffer needs. Additionally, it holds information about the queue family on which the allocated command buffer will be used. [5]

After the command buffer is created commands, that represent the work to be done, are recorded to it. In case of recording a command buffer for rendering frames all necessary resources, as vertex buffers, index buffers, descriptor sets and draw commands, are bound and recorded to it. [5]

When finished recording commands, it gets submitted to a specified queue, that fulfills the demands the of the command buffer. [5] Figure 3.9 gives an overview of how a command buffer looks like for rendering an image.

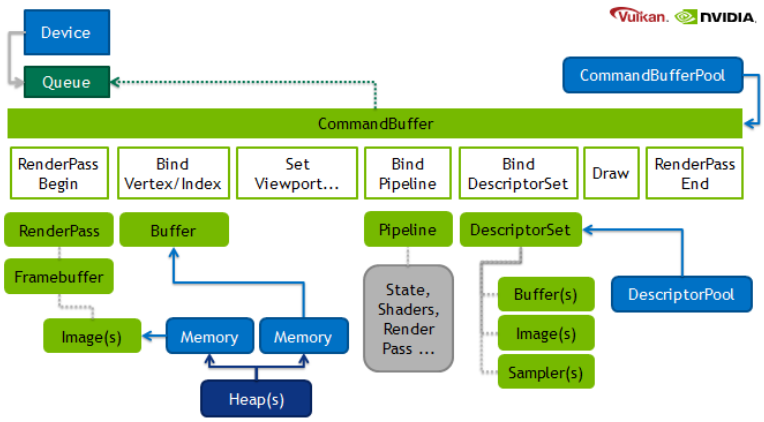


Figure 3.9: Overview of a command buffer for rendering. (taken from [9])

3.4.7 Synchronization Primitives

Vulkan is designed with an explicit synchronization of executions on the GPU, that the developer needs to take care of. Whenever tasks of a command buffer are submitted to a queue, the GPU starts working on them asynchronously. Therefore, the CPU part of the application does not wait until the GPU is done executing. As a result, Vulkan uses two different synchronization primitives that help synchronizing tasks. In the following these two are introduced regarding to their usage. [39]

3.4.7.1 Semaphores

Semaphores are used to identify when a the commands of a command buffer are finished processing. They can be passed on as parameters when submitting work to a queue. They can be either in a unsignaled or signaled state. A signaled state means, that all commands of the submitted command buffer are done. The goal of using semaphores is to synchronize tasks on the GPU and to guarantee an implicit memory usage for each command buffer. [41]

3.4.7.2 Fences

Fences, as contrast, are used to synchronize tasks between CPU and GPU. As same as semaphores, they are attached to the submission of a command buffer to a queue and have an unsignaled and signaled state. The application can check on a fence status, by using the function *vkGetFenceStatus* to receive the fence's state or *vkWaitForFences* to block the CPU until all tasks are done and the fence gets signaled. [41]

4 Rendering Methods

The development of terrain rendering methods and algorithm is a research field, which researchers are occupying themselves with over the last decades. With the steadily evolution of new hardware features, existing methods and algorithms must be adapted or even new ones must be developed in order to take full advantage of modern hardware capabilities. Therefore, in this chapter important rendering methods and algorithms which were developed over the course of time will be introduced in chronological order. The goal is to introduce different terrain rendering approaches, that will build the basis, when it comes to choosing a suitable method for this thesis's renderer. Furthermore, this chapter is not intended to describe each technique in detail, but to give a brief overview of their basic functionality, as well as their up and downsides.

4.1 Real-time Optimally Adapting Meshes

The real-time optimally adapting mesh (ROAM) terrain rendering method was introduced by M. Duchaineau et al. in 1997. ROAM uses dynamic, view-independent triangle meshes and texture maps to achieve optimized rendering quality, as well two priority queues for split and merging operations. The algorithm works with a preprocessing stage and a runtime stage, that performs updates on the triangulated mesh by splitting and merging triangles. During the preprocessing stage a nested, view-independent error bound for a triangle bintree is created. This error bound ensures constant visual results from different views, as the triangulation of the terrain is dynamically modified during runtime. The result of the algorithm is an optimal RTIN, that is updated from frame to frame, to ensure an optimal triangulated mesh referring to its error bound. In addition, multiple performance optimizations like view-frustum culling and progressive optimizations for the mesh are explained. [14]

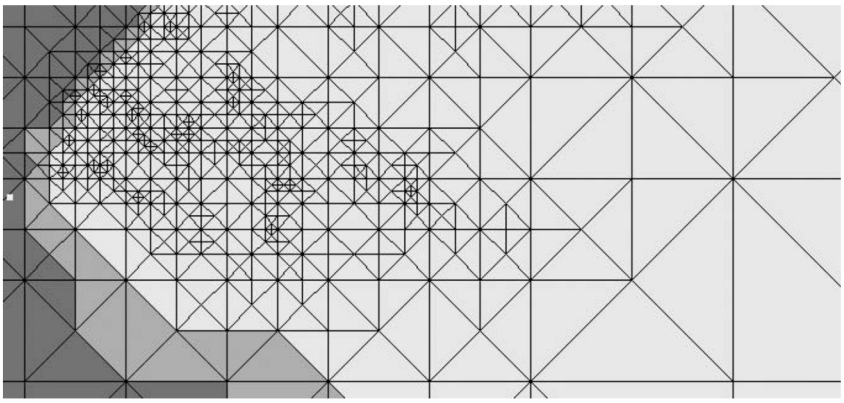


Figure 4.1: Example of a triangulation of the ROAM algorithm. (taken from [14])

Continuous level of detail (CLOD) algorithms, like ROAM, are capable of creating an optimal view-dependent triangulation of terrain by using a simple, but powerful mesh refinement process. However

these algorithms are not as suitable for modern GPU hardware, due to their high CPU overhead. [12] ROAM uses CPU computations for the mesh refinement and merging, what causes many data uploads to the GPU. A frequent upload of an optimal CPU processed triangulation can be more expensive than rendering a less optimal terrain mesh which vertex data is already available for the GPU. Since the availability of more powerful consumer GPUs, optimal render methods should minimize their CPU preprocessing phases and especially their amount of data transfers from the CPU to the GPU. [49]

4.2 Chunked LOD

Chunked LOD is a MLOD algorithm introduced by Thatcher Ulrich, that uses a quadtree structure to organize the blocks. The blocks of the quadtree in this method are considered as chunks and consist of independent static meshes from a preprocessing phase. For the creation of each chunk's mesh is a high-detail reference mesh taken into account. The structure of the quadtree is similiar as to the in section 2.4.3 described structure, where the root represents the whole object in a coarse low-level mesh, while the childnodes refine the object into more detailed and smaller sized chunks. Ulrich assigns a bounding volume and maximum geometric error to each chunk, where chunks at the same level are given the same geometric error. A screen-space error can be derived from these two values, to determine during run-time whether a chunk is suitable for the current desired visual accuracy. [49]

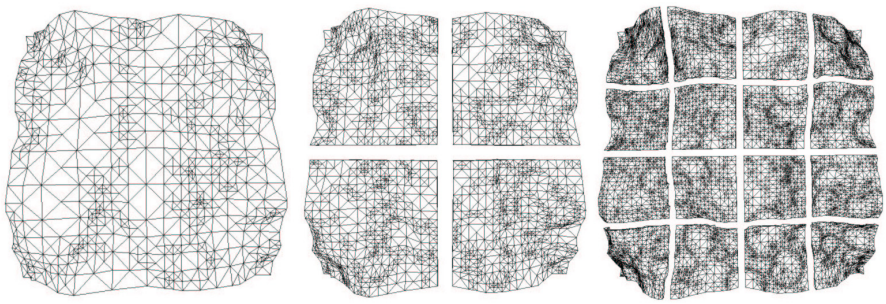


Figure 4.2: Three level of Chunked LOD tree. (taken from [49])

Furthermore, Ulrich deals with the two common visual issues cracks and popping. For the former he introduces a crack filling solutions, that uses a vertical skirt pointing downwards at each of the chunks' edges to hide visible holes between two chunks. For the latter he uses a morphing algorithm to achieve a smoother transition whenever a chunk is split into its child chunks or child chunks are merged to their parent chunk. [49]

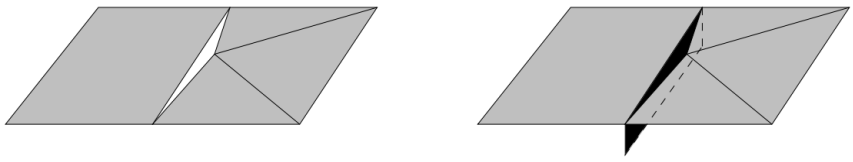
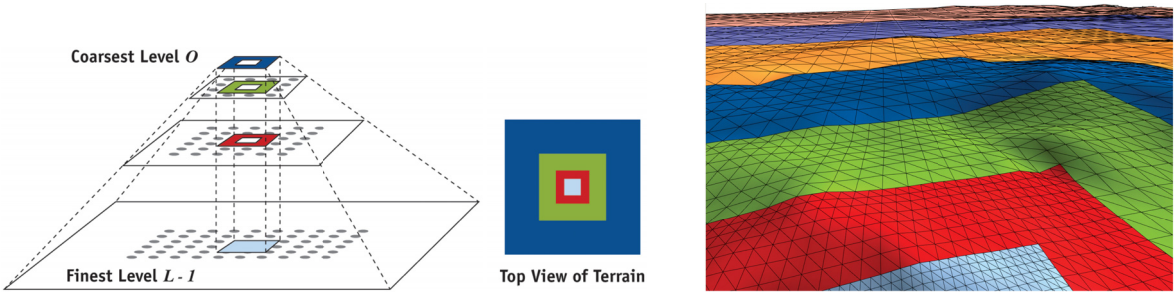


Figure 4.3: Visual demonstration of the crack filling method. A vertical skirt (black rectangle) is used to fill cracks between two chunks. (taken from [49])

Ulrich claims his rendering method achieves a low CPU overhead, due to already preprocessed refined meshes. Therefore, this rendering method can be regarded as a GPU friendly algorithm, that puts through most of the work to the GPU at runtime. [49],[12]

4.3 Geometry Clipmaps

One of the most prominent terrain rendering method is Frank Losasso and Hugues Hoppe's introduced geometry clipmap rendering method. Their technique is designed to handle the rendering of large sized heightmaps. The geometry clipmap represents the terrain in a set of nested regular grids that is centered around the viewer's position. This structure can be regarded as a pyramid, that consists m levels, where each level represents one of the nested regular grids. Each grid contains a $n \times n$ sized array of vertices, while the resolution increases successively to the power of two from a coarser to the next finer level. As the viewpoint moves across the world, the clipmap shifts along with the viewpoint and refills the clipmap with updated data. [30]



(a) Visual representation of a geometry clipmap structure. (taken from [30])

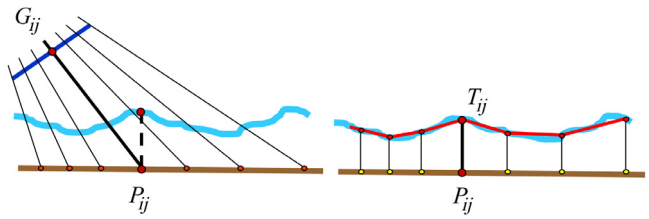
(b) Result of rendered terrain with geometry clipmaps. (taken from [30])

Figure 4.4: Visualizes parts of geometry clipmap algorithm.

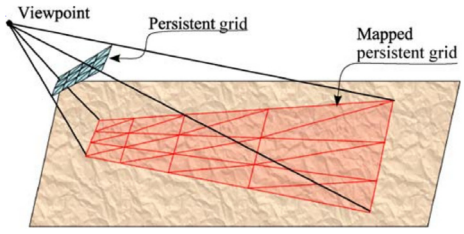
Compared to other rendering algorithms, geometry clipmaps ignore the local surface geometry of terrain and therefore do not consider screen-space errors for the triangulation of the mesh. The only refinement criteria taken into account is the distance to the viewer. This leads to a regular grid triangulation at each level of the clipmap structure. In addition, to avoid an abrupt transition between different levels, Losasso and Hoppe are using morphing algorithms to secure a smoother visual result. [30]
Rendering terrain using this approach ensures an optimal rendering throughput, as a result of storing the terrain data with a vertex buffer directly in GPU's video memory. [30] However the geometry clipmap is a complex structure for the representation of terrain and as a regular grid for the triangulation is used, this method may cause an unnecessary amount of triangles for certain terrain geometries. [12]

4.4 Persistent Grid Mapping

Persistent grid mapping (PGM) is a terrain rendering method developed by Yotam Livny et al. and uses advantages of ray-tracing and mesh-rendering, that leads to a different approach than previous mentioned rendering methods. The algorithm uses a sampled grid that is mapped from the viewport on a base terrain surface. Since the resolution of the viewport is usually not resized, the size of the sampled grid stays unmodified as well. Therefore the grid is referred to as a persistent grid. The mapping of the persistent grid is done by shooting rays from the viewport through every grid's vertex and computing intersection points of a ray on the base terrain surface. It is also important to mention that the base terrain surface does not include the height information of the terrain. Hence, each vertex's height value of the mapped persistent grid is equal to zero. The height values for each vertex are looked up by using its x and y coordinates from an elevation map after the mapping is done. [29] Figure 4.7 illustrates the described process of mapping and assigning the height values.



(a) Illustration of mapping the persistent grid on terrain base surface(left) and assigning height value to a vertex (right). G_{ij} represents vertex on persistent grid, P_{ij} corresponding mapped point and T_{ij} final vertex position with assigned height value. (taken from [29])



(b) Visualization of persistent grid mapping. (taken from [29])

Figure 4.5: Visualizations of mapping and assigning of height values.

One major issue of the described procedure is that in certain scenarios terrain geometry which should be visible to the viewer is not rendered. This is the case whenever peaks of mountains are close to camera, but are no part of the projected persistent grid. To solve this issue an additional sampling camera is used for sampling terrain data and generating the mesh. The sampling camera uses a different viewing angle in order to consider terrain areas close to the camera, that may include visible parts for the actual viewing camera. [29]

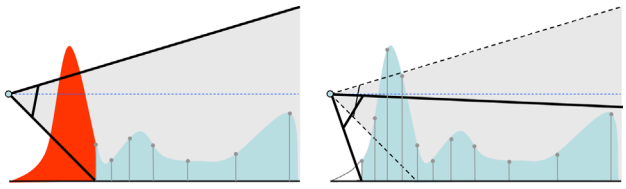


Figure 4.6: Shows camera restriction (left) and the use of of sampling camera (right). (taken from [29])

Linvy et al. claim that the PGM algorithm is capable of rendering terrain with stable framerates due to rendering a mesh of the same complexity in each frame. The algorithm also ensures crack free mesh with smooth LOD transitions without the need of any preprocessing stages. In addition, no extra culling algorithms are necessary, because of the grid covering only the visible part of the frustum. [29]

4.5 RASTeR

RASTeR is a continuous LOD rendering method introduced by Jonas Bösch et al., that puts its focus on a patch-oriented multiresolution triangulation and rendering, instead of individual vertices or triangles. The rendering technique relies on the introduced paired mutltiresolution tree structure. The goal of the data structure is to separate LOD selection and rendering, and uses two new concepts K-patches and M-blocks. [7]
A K-Patch represents a triangle having K numbers of vertices on each edge, and has one of eight orientations. Through selectively splitting a K-patch in smaller patches, an adaptive LOD can be achieved. The refinement of a K-Patch is represented in a bintree, which is referred to as a meta bintree, where each node represents a K-patch. Using these K-patches for mesh triangulation, allows an adaptive LOD triangulation based on the viewing distance. [7]

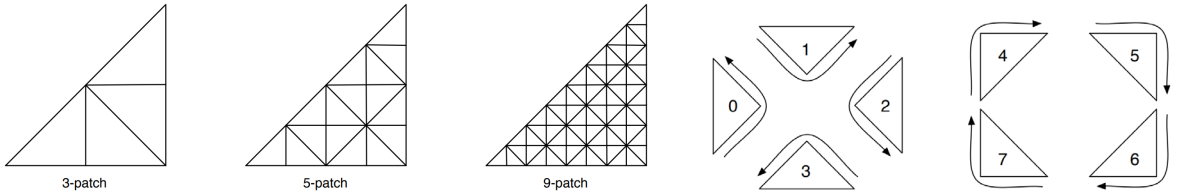


Figure 4.7: Illustration, where (a) shows K-patches of different sizes and (b) visualizes the eight orientations of K-patches. (both taken from [7])

A M-block is a square sized block, that contains a regular grid of height data. The size among all M-blocks is equal to the size of $M \times M$ vertices. M-blocks can also be refined to child M-blocks. This refinement is organized in a quadtree, where each node represents a M-block.
Additionally, a meta bintree node holds a pointer to a corresponding quadtree node, to connect both trees with each other. The relationship between K-patches and M-blocks ensures a decoupling of LOD selection and actual rendering, allowing a separate management of the mesh geometry and the terrain's height data. [7]

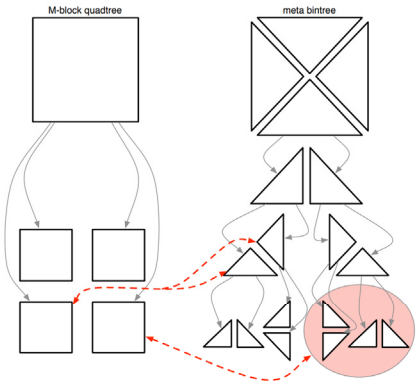


Figure 4.8: Shows a M-Block quadtree and a meta bintree with connections between K-patches and M-blocks. (taken from [7])

4.6 Hardware Tessellation Rendering

Goanghung Kim et al. introduced a rendering method based on hardware tessellation. In this method LOD management is performed and dynamically controlled by the tessellation stage of the rendering pipeline. The tessellation stage uses flat quadliteral patches as input primitives on which tessellation is performed. The tessellation control shader is used for determining the LOD for each input primitive, by calculating tessellation levels. The calculation of the LOD is done by an algorithm based on the distance between the viewpoint and the quadliteral patch.
The tessellation evaluation shader is used for generating the actual terrain geometry. This is done by assigning a height value to each vertex of the tessellated patch. The necessary height values are supplied to the tesselleation evaluation shader by a heightmap. [22]

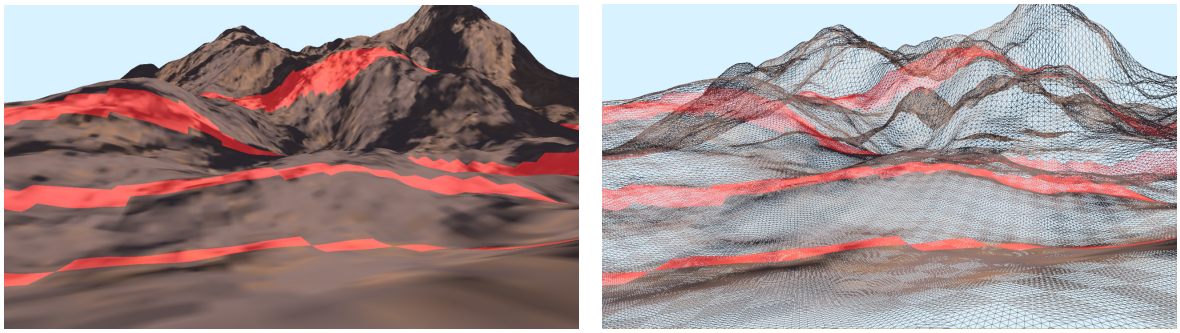


Figure 4.9: Results of the hardware tessellation rendering method. The red stripes mark transition areas. (both taken from [22])

Additionally, Kim et al. are trying to solve the common issues of popping and cracks. For the former they define a transition area at the end of every LOD interval, as it can be seen in Figure 4.10. In this area a morphing process on each patch is applied to ensure a smooth transition between different LOD. For solving crack effects, the tessellation evaluation shader offsets T-shape vertices at the edge of a patch. The authors claim determining T-shape vertices can be done with just a small amount of extra computations. [22]

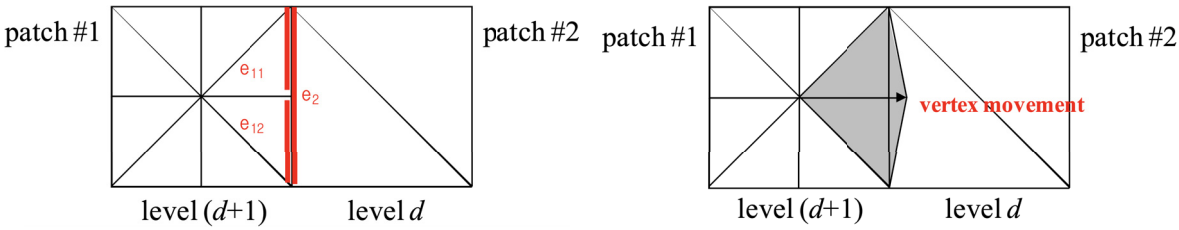


Figure 4.10: Figure on the left shows two patches where a crack can occur. Figure on the right illustrates offset of vertex to prevent a crack. (both taken from [22])

Kim et al. introduced a rendering method that offers a high throughput on the GPU and a low CPU overhead, since the LOD management is completely performed on the GPU. [22]

5 Design Decisions

With having knowledge of a technical background of terrain rendering and the rendering pipeline, as well of different rendering methods, in this chapter requirements on the terrain renderer which should be developed will be explained. Afterwards, the introduced rendering methods will be evaluated to find a suitable solution for the implementation. Finally, an overview of the implementation's method will be given and explained.

5.1 Requirements

As a base for the terrain renderer, the graphic API Vulkan, that was described in chapter 3 should be used. The reason why this decision was made is to have benchmark data that can be compared to an already existing renderer with similar visual results based on OpenGL. Since Vulkan is a modern and platform independent graphic API which claims to grant better performance due to modern concepts, better performance results should be expected compared to the OpenGL implementation.

For the required real-world height data DEM files, where each file represents an area with the dimensions of 1km x 1km, will be used. For texturing the terrain digital orthophotos should be used, where each one corresponds to an area of the DEM files.

Furthermore, the terrain renderer should be executable on embedded devices, while being capable of delivering at least one frame every *20*ms. The challange on this requirement is due to the hardware capability of embedded devices. Embedded devices are usually equipped with simpler and less advanced hardware features.

Lastly, the terrain renderer should work off-screen. Therefore, the rendered images should not be displayed on a window surface, but should be transferred to a location, where other processes can access them for further processing.

5.2 Evaluation of Existing Methods

In this section, the introduced rendering methods from chapter 4 will be evaluated. The goal is to find a well suited solution for the aforementioned requirements, in special regard to embedded systems. It should be also mentioned, that the focus is put on rendering algorithms with the least amount of CPU overhead, in order to take full advantage of the GPU's performance.

5.2.1 CPU vs GPU

The introduced ROAM algorithm is the oldest of the mentioned methods and uses exclusively a CPU based algorithm for the LOD management and mesh refinement process. When regarding newer approaches, a change from CPU based to GPU based algorithms is noticeable. This is due to the drastically improvement of GPUs. Since they became more powerful, CPU based rendering algorithms have become obsolete. Therefore, CPU based rendering methods, like ROAM, will not be further considered and instead GPU based rendering methods will be focused.

5.2.2 Level of Detail Management

The LOD management is crucial for the performance of a rendering method. The introduced rendering methods are using different approaches to achieve an optimal LOD management. Some of them, like chunked LOD , rely on pre-computed simplifications of the meshes, where the application decides during runtime which simplification is most suitable for the current situation. However, having multiple versions of the same terrain area is not the best choice for data sets of large areas. Furthermore, geometry clipmaps and RASTeR, both introduced interesting and efficient LOD management approaches. But due to their complex structures that are used to store, update and render the meshes they will be not considered in this thesis's renderer. Lastly, there are persistent grid mapping and hardware tessellation rendering as possible rendering methods left. Both of them use efficient LOD management strategies, that offer good performance results. In regard of the second requirement, mentioned in the previous section, the persistent grid mapping method might be not as suitable. Since the mapping of the grid does not follow any pattern, its position is always random, resulting in complicated accessing of the height values from different height maps. As contrast, a hardware tessellation approach seems to be a good choice, since the LOD management takes place completely on the GPU and therefore reduces the workload of the CPU.

5.3 Method Overview

Due to embedded devices having not as much performance as regular consumer hardware, the selected method aims to avoid a complex pre-processing stage in the rendering loop, especially on the CPU side. As in the evaluation could be seen, that a hardware tessellation approach seems to be a good choice for the implementation. With that no complex LOD selection on the CPU has to take place, since the tessellation shader can determine the triangulation of the terrain model during runtime. Therefore the selected method will use a hardware tessellation approach with the general ideas of the method described in section 4.6.

The method overall consists of three different processes as it can be seen in figure 5.1. The initialization process is used for loading and creating all necessary data and data structures based on the initial viewpoint. The render process and chunkmap update process are concurrently executed in different threads after the initialization process has finished. Both of them represent a loop that constantly repeats until the application is stopped. Each process will be described in detail in the next sections.

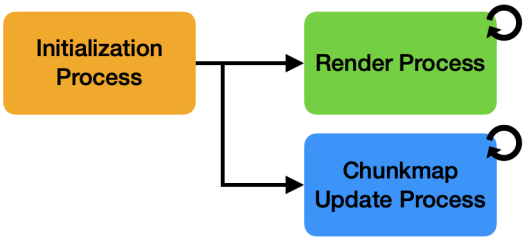


Figure 5.1: Overview of the renderer's processes.

5.3.1 Data Structure

For the organization of all necessary data for the render process a chunkmap will be used. The chunkmap consists of multiple chunks, where each chunk represents an area of the terrain. The size among all chunks is equal and depends on the dimensions of the area that each chunk represents. Each area of a chunk in this case is defined by a DEM tile. Since the dimensions of a DEM tile is 1km x 1km, a chunk is 1000 x 1000 in size in terms of the world coordinate system. The amount of chunks in a chunkmap is calculated based on the view distance and the chunk width. The view distance can be set to a desired value, but remains constant while the renderer runs. Equation 5.1 shows how the chunk count is calculated.

$$chunkCount = (\lceil \frac{viewDistance}{chunkWidth} \rceil * 2 + 1)^2 \tag{5.1}$$

The chunkmap is always centered around the view point and the calculated chunk count ensures that the chunkmap is large enough for the viewer to look at any direction with the defined view distance. Figure 5.2 shows an example of the structure of a chunkmap to get a better idea of how it is build up.

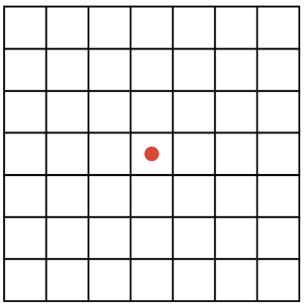


Figure 5.2: Show chunkmap with a viewdistance of 3000. Red dot marks the view point.

5.3.2 Initialization Process

The initialization process is used to setup the initial chunkmap with all necessary data, such as creating a flat model, which will be explained shortly, loading height map data and texture images for each chunk.

Figure 5.3: Overview of the initialization process.

5.3.2.1 Flat Model Creation

A flat model represents a plane surface for each chunk with all height values set to zero. It consists of stripes with squares, where each square is divided in two isosceles triangles. The amount of squares in each stripe depends on a value m that is set for the width of each square. Figure 5.4 shows how a flat model looks like.

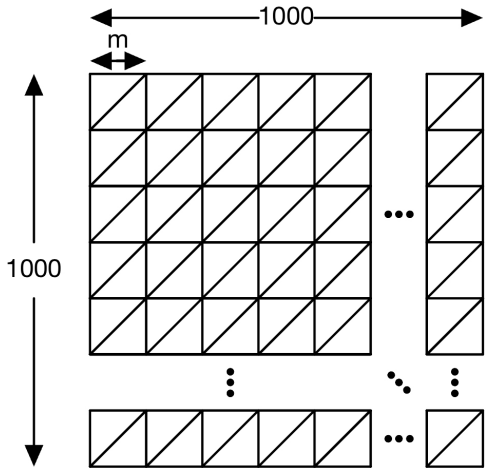


Figure 5.4: Shows structure of a flat model. (adapted from [25])

The purpose of a flat model is to have a plane of triangles that serves as a base geomerty, on which the tessellation shader performs the tessellation. Therefore, it is crucial to use a suitable value for the width m, in order to generate a base geometry that is not too detailed. If the model is too detailed, the tessellation shader has not enough possibilities for a proper LOD management. However, when the flat model is too coarse, the tessellation shader has to compute a larger amount of new vertices, which ends up disturbing the performance. [25] The generated vertices of all chunks' flat models are put together in a single vertex buffer. Furthermore an index buffer is used to keep the amount of vertices in the vertex buffer as small as possible.

5.3.3 Render Process

The render process represents the core part of the terrain renderer. In here the actual rendering of the frames takes place and it can be regarded as sequence of five steps which are necessary to do so. Therefore, in this section the steps of the render process will be presented and explained with regard to their tasks.

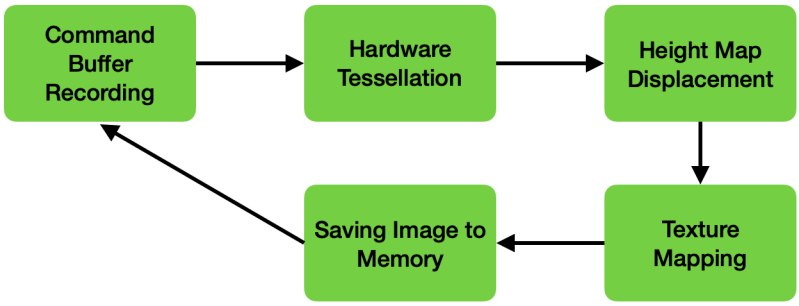


Figure 5.5: Overview of the render process.

5.3.3.1 Command Buffer Recording

The command buffer recording phase determines which parts of the chunkmap needs to be sent to the rendering pipeline. It uses frustum culling, to check whether a chunk is within the camera's field of view, and therefore only records draw commands for chunks that are actually visible. This is a useful performance optimization, because it can drastically reduce the amount of chunks which will be rendered and tessellated. Figure 5.6 visualizes the effect of frustum culling. For this example a chunkmap which is based on a view distance of 3000.0 is used. As it can be seen the amount of chunks which are considered for tessellation and rendering can be reduced from *49* to *10* chunks, which almost reduces the amount of chunks by factor *5*. Of course this is just a theoretical scenario. The actual amount of chunks which should be rendered depends on the field of view angle, as well as the view direction of the camera. Nevertheless, a performance increase from using this method can be expected.

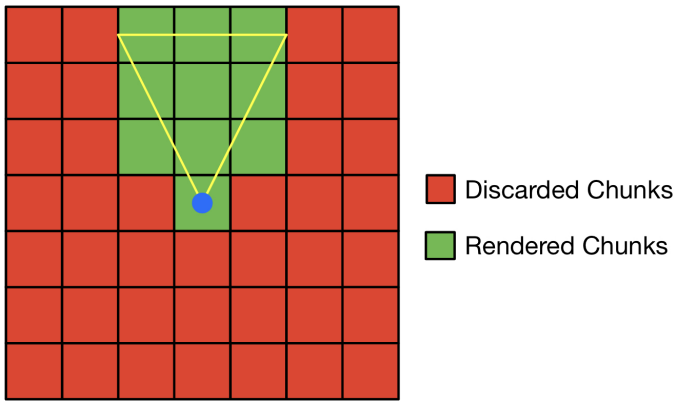


Figure 5.6: Shows impact of frustum culling.

5.3.3.2 Hardware Tessellation

The hardware tessellation phase can be regarded as a LOD management process. The main part of this phase takes place in the tessellation control shader (TCS). The TCS computes based on a screen space error the outer and inner tessellation levels for each triangle patch of the flat model. The computed tessellation levels determine how detailed a triangle patch will be tessellated and rendered. A detailed explanation of this process will be explained later on in chapter 6.

5.3.3.3 Height Map Displacement

After the determining of the tessellation levels in the TCS and generation of new vertices by the tessellation primitive generation, the tessellation evaluation shader (TES) takes on the displacement of each vertex's height value. This is done by accessing the corresponding height map with each vertex's defined texture coordinates and adding the returned value to the height coordinate of the vertices. By finishing the height map displacement phase, the final mesh of the terrain's geometry is generated.

5.3.3.4 Texture Mapping

The final generated terrain mesh now needs to get textured with the digital orthophotos. Texture mapping is performed in the fragment shader by simply accessing the corresponding texture that was created in the initialization process. For receiving the correct part of the texture the same texture coordinates of each vertex that were already used for acquiring the necessary height data from a height map.

5.3.3.5 Saving Image to Memory

After an image was rendered it is stored to a framebuffer object, which image attachements are usually stored in device-local memory. Since the rendered images are supposed to be accessible by other processes it is necessary to transfer them to a host visible memory location.

5.3.4 Chunkmap Update Process

With now having explained how the render process works, there is one aspect still open that needs to get taken care of. This one is the update process for the chunkmap. So far only the area which was defined by the initialization process can be rendered. But when the camera moves through the world, new chunks have to get loaded and added to the chunkmap, while chunks which are not in range anymore have to get discarded from the chunkmap. Figure 5.7 visualizes this process and gives a basic idea of what needs to get done.

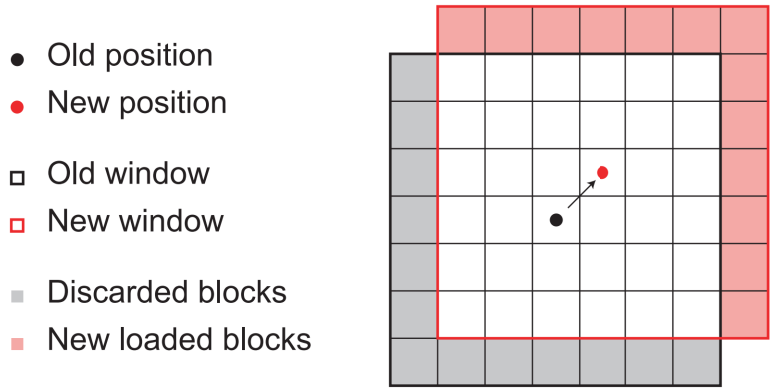


Figure 5.7: General idea of the update process. (taken from [26])

The chunkmap update process runs concurrently to the render process in a separate thread and can be split up into three different phases.

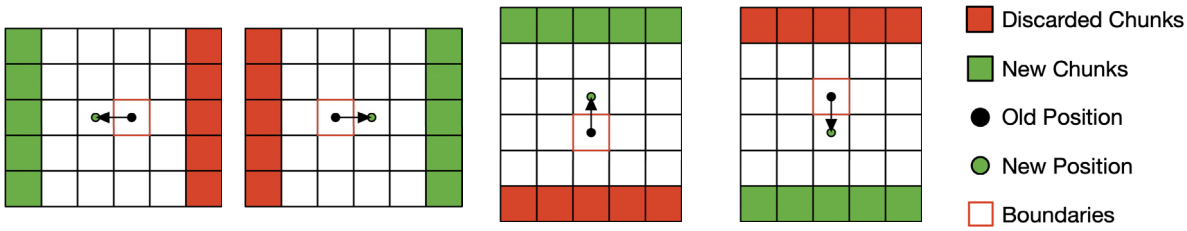
Figure 5.8: Overview of the chunkmap update process sequence.

5.3.4.1 Position Checking

The process starts with the phase of position checking. To determine whether and especially what new data is necessary to load for updating the chunkmap, four boundaries are defined. These boundaries are defined by the position of the center chunk. The smallest x coordinate defines the left boundary, the largest x coordinate the right boundary, the smallest y coordinate the lower boundary and the largest y coordinate the upper boundary. If one of these boundaries is passed, the process will continue with the next phase, otherwise it will continue to check until a boundary is passed.

5.3.4.2 Data Loading and Updating

Depending on which boundary is passed, the data loading phase determines the new chunks which will be added to chunkmap and loads the necessary height maps and textures. Figure 5.9 illustrates based on the passed boundary, which chunks need to get discarded, and which chunks will be added to the chunkmap. Furthermore, it creates new flat models for the new chunks. The newly loaded data and generated flat models will be buffered in a temporary buffer before finally added to the chunkmap. The reason for the buffering of the data is, that the first two phases can run concurrently to the render process without the need of any synchronization. The data updating phase, however, needs to be synchronized with the render phase, because the data of the chunkmap is constantly used when rendering frames. In order to synchronize the two processes, the in section 3.4.7 described fences for synchronization are used. Whenever the GPU finishes rendering a frame, it puts the fence in the signalized state. Then the data updating phase knows when it can safely update the chunkmap by discarding and adding chunks.



(a) Passing of left boundary. (b) Passing of right boundary. (c) Passing of upper boundary. (d) Passing of lower boundary.

Figure 5.9: Shows the four cases of boundary passing.

6 Implementation

In the last chapter the methods and ideas of the terrain renderer were explained. This chapter will deal and show the core parts of the renderer's implementation. These include the chunkmap's data structure, frustum culling and the tessellation stage.

6.1 Chunkmap Data Structure

```
1  struct ChunkMap {
2      vector<Image> heightMaps;
3
4      vector<Image> textureImages;
5
6      vector<vector<Vertex>> vertices;
7
8      vector<uint32_t> indices;
9
10     vector<MeshIndexInfo> meshindexinfos;
11
12     vector<BoundingBox> boundingBoxes;
13
14     Boundary boundary;
15
16     int centerChunk[2];
17 };
```

Code 6.1: Data structure of chunkmap

The data structure of the chunkmap can be seen in code 6.1. It basically holds the necessary data for each chunk, which consists of the height map, texture images and the flat model vertices. The height maps and texture images are each stored in an one-dimensonial vector of the type *Image*. The *Image* type represents a struct, that contains the three data types VkImage, VkDeviceMemory and VkImageView, which are necessary for using images in Vulkan. The vertices of each chunk's flat model are stored in a vector of the type *Vertex*. The *Vertex* type, again, is a struct, that contains position and texture coordinates of the vertex. These *Vertex* vectors are united in a vector, where each field of the vector corresponds to a chunk. An indices vector is used, to store all indices, that are used later on in an index buffer to be able to use indexed draw commands. By using the *MeshIndexInfo* type, the indices that correspond to a chunk's flat model can be accessed. This is done by a start index, to acquire the first flat model's index from the indices vector and the amount of indices which are used for each flat model. The vector of type *BoundingBox* stores the position of the four corner vertices of each chunk and is used as information for frustum culling. The boundary attribute, as expected, holds the current boundaries, that are used for updating the chunkmap, as described in section 5.3.4. Lastly, the centerChunk array contains information about the center chunk's file names, on which the loading of new height maps and textures depends.

```
struct Image {
    VkImage image;
    VkDeviceMemory imageMemory;
    VkImageView imageView;
};

struct Vertex {
    glm::vec3 pos;
    glm::vec2 texCoord;
    ...
};

struct MeshIndexInfos {
    uint32_t startIndex;
    uint32_t numIndices;
};

struct Boundary {
    int upperBoundary;
    int lowerBoundary;
    int leftBoundary;
    int rightBoundary;
};

struct BoundingBox {
    glm::vec3 bottomLeft;
    glm::vec3 bottomRight;
    glm::vec3 topLeft;
    glm::vec3 topRight;
};
```

Code 6.2: Shows used data types in chunkmap.

6.2 Frustum Culling

Frustum culling is used for culling all chunks, that are not visible to the viewer. It is performed on the CPU side before the command buffer gets filled with draw commands. Only chunks, that pass the visibility test will be recorded to the command buffer. Before frustum culling on a chunk can be performed, the current viewing planes have to be updated. For those needs the function, that can be seen in Code 6.3, is used. The function receives a view-projection-matrix that is the product of the view and projection matrix, as they are used in the uniform buffer, and a reference to an array that holds the six planes. Before the planes can be calculated from the view-projection-matrix, it has to be transposed in order to switch the matrix from a column-major to a row-major order. This is necessary for an easier access of the matrix's components, that are used to calculate the frustum planes.

```
void getFrustumPlanes(glm::mat4 viewproj, glm::vec4* planes) {
    viewproj = glm::transpose(viewproj);
    planes[0] = glm::vec4(viewproj[3] + viewproj[0]); // left
    planes[1] = glm::vec4(viewproj[3] - viewproj[0]); // right
    planes[2] = glm::vec4(viewproj[3] + viewproj[1]); // bottom
    planes[3] = glm::vec4(viewproj[3] - viewproj[1]); // top
    planes[4] = glm::vec4(viewproj[3] + viewproj[2]); // near
    planes[5] = glm::vec4(viewproj[3] - viewproj[2]); /* far*/ }
```

Code 6.3: Function for calculating frustum planes. (adapted from [40])

During the command buffer recording phase, for each chunk of the chunkmap is checked whether it is visible or not. This is done by using the the function *isChunkVisible*, that can be seen in code 6.4. The function takes the afore calculated planes and a *BoundingBox* of the chunk to be checked as parameters. The *BoundingBox* of a chunk is a struct, that holds the four corner coordinate of the chunk. These coordinates also include the actual height that they would have in the final terrain mesh. The function checks for every plane whether the chunk's corners are behind or in front of a plane. This check is performed by calculating the dot product between the plane's vector and the each of the chunk's corner. A result less than zero means, that a corner is behind a plane. In case of all four corners of a chunk being behind one of the planes, means that the chunk is not visible and will not be recorded to the command buffer.

```
1  bool isChunkVisible(glm::vec4* frustumPlanes, const BoundingBox& box) {
2    for ( int i = 0; i < 6; i++ ) {
3      int r = 0;
4      r += (glm::dot(frustumPlanes[i], glm::vec4(box.bottomLeft, 1.0f)) < 0.0) ?
       1 : 0;
5      r += (glm::dot(frustumPlanes[i], glm::vec4(box.bottomRight, 1.0f)) < 0.0) ?
        1 : 0;
6      r += (glm::dot(frustumPlanes[i], glm::vec4(box.topLeft, 1.0f)) < 0.0) ? 1 :
        0;
7      r += (glm::dot(frustumPlanes[i], glm::vec4(box.topRight, 1.0f)) < 0.0) ? 1
       : 0;
8      if (r == 4) {
9        return false;
10     }
11   }
12   return true;
13 }
```

Code 6.4: Function for visibility check. (adapted from [40])

When tests for this frustum culling approach were made, at certain view directions some visible chunks were not considered as visible, what led to black areas in the terrain. This only happened to some chunks that were at the edge of the screen. To solve this issue the field of view angle of the projection matrix gets increased a little bit when calculating the frustum's planes, but stays the same when supplying it to the shaders for rendering. With that the frustum for the visibility check is increased slightly and avoids false visibility consideration of chunks.

6.3 Tessellation Stage

As already mentioned, the tessellation stage represents an important part for the renderer. The tessellation control shader (TCS) is used to determine based on a screen space error how detailed a triangle patch gets tessellated and therefore states an important performance optimization. The tessellation evaluation shader (TES) on the other hand is used to assign height values to each already existing and newly generated vertex. Therefore, in this section the functioning of both shaders will be explained.

6.3.1 Tessellation Control Shader

At first of all, it useful to have a look at the input and output values of the TCS. The first input value is *vert_texcoord*, which are texture coordinates. These texture coordinates are passed on from the previous stage, which is in this case the vertex shader. Furthermore, the TCS accesses the uniform buffer object from a bound descriptor set, that contains transformation matrices, as well as a tessellation factor, viewport dimensions and another value called *tessellatedEdgeSize* which will be used for computing the tessellation levels. As an userdefined output, texture coordinates will be passed on to the next stage. Furthermore, line 12 in Code 6.5 defines, that the TCS is configured on output patches with three vertices, which basically means triangles. Besides the userdefined input and output variables, GLSL uses built-in variables at each shader stage. In case of the TCS the used built-in input variables are *gl_InvocationID* and *gl_in*. The former variable is for indexing the vertices of the input patch. The latter represents a structure for each vertex of the patch, that includes the variable *gl_Position* holding the position of the vertex. For the built-in output variables *gl_out*, *gl_TessLevelOuter* and *gl_TessLevelInner* are used. *gl_out* holds the same information as *gl_in*, while *gl_TessLevelOuter* and *gl_TessLevelInner* are used to pass on the computed tessellation levels to the primitive generation.

```
1  layout(location = 0) in vec2 vert_texcoord[];
2  layout(binding = 0) uniform UniformBuffer {
3      mat4 model;
4      mat4 view;
5      mat4 proj;
6      float tessellationFactor;
7      vec2 viewportDim;
8      float tessellatedEdgeSize;
9  } ubo;
10
11 layout(location = 0) out vec2 tesc_texcoord[];
12 layout(vertices = 3) out;
```

Code 6.5: Input and output of the tessellation controll shader.

The function *screenSpaceTessFactor* is used for calculating the tessellation of the triangles edges. It takes two vertices of an edge as an input and calculates the distance between both points in screen space. Before the distance between those vertices can be calculated, the vertices' positions have to get transformed from world space coordinates to screen space coordinates. The calculated distance between the vertices' position in screen space is then adjusted by the two factors *tessellatedEdgeSize* and *tessellationFactor* to calculate the tessellation level. Lastly, it has to be ensured that the tessellation level is in range of permissible values. This range is defined for values in [1.0, 64.0]. In order to achieve that the tessellation levels are in this range, the built-in function *clamp* is used. By using

this method for the calculation of the tessellation levels that the tessellation of the triangle patches is according to a screen space error securing an optimal level of detail selection.

```
float screenSpaceTessFactor(vec4 p0, vec4 p1) {
    vec4 midPoint = 0.5 * (p0 + p1);

    float radius = distance(p0, p1) / 2.0;

    vec4 v0 = ubo.view * midPoint;

    vec4 clip0 = (ubo.proj * (v0 - vec4(radius, vec3(0.0))));
    vec4 clip1 = (ubo.proj * (v0 + vec4(radius, vec3(0.0))));

    clip0 /= clip0.w;
    clip1 /= clip1.w;

    clip0.xy *= ubo.viewportDim;
    clip1.xy *= ubo.viewportDim;

    return clamp(distance(clip0, clip1) / ubo.tessellatedEdgeSize * ubo.
    tessellationFactor, 1.0, 64.0);
}
```

Code 6.6: Function for determining tessellation level. (adapted from [55])

The main function of the TCS uses the *screenSpaceTessFactor* function to assign a tessellation level to each of the three outer tessellation levels of the built-in *gl_TessLevelOuter* variable. For the inner tessellation level of *gl_TessLevelInner* the built in function *mix* is used. This function performs a linear interpolation between two values by using a factor to weight between them. After the tessellation levels are computed, the TCS passes them on to the next stage, together with the position of the triangle's vertices and texture coordinates, by assigning them to the corresponding output variables.

```
void main() {
    if (gl_InvocationID == 0) {
        gl_TessLevelOuter[0] = screenSpaceTessFactor(gl_in[1].gl_Position,
    gl_in[2].gl_Position);
        gl_TessLevelOuter[1] = screenSpaceTessFactor(gl_in[2].gl_Position,
    gl_in[0].gl_Position);
        gl_TessLevelOuter[2] = screenSpaceTessFactor(gl_in[0].gl_Position,
    gl_in[1].gl_Position);
        gl_TessLevelInner[0] = mix(mix(gl_TessLevelOuter[0], gl_TessLevelOuter
    [1], 0.5), gl_TessLevelOuter[2], 0.5);
    }

    gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
    tesc_texcoord[gl_InvocationID] = vert_texcoord[gl_InvocationID];
}
```

Code 6.7: Main function of tessellation controll shader. (adapted from [55])

6.3.2 Tessellation Evaluation Shader

As in the tessellation evaluation shader, it is useful to have a look at the input and output variables of the tessellation evaluation shader (TES). As input variables, the TES receives texture coordinates from the TCS. It also takes the uniform buffer and the height map's texture sampler from the descriptor set. Furthermore, the built-in *gl_in* and *gl_TessCoord* variables are used. The former was already described in the last section, while the latter is a location vector with the values x,y and z assigned with float values in range of [0, 1]. As output, the TES passes on texture coordinates to next stage, together with the built-in variable *gl_Position*, that represents the position of the in clip space. Additionally, line 1 in Code 6.8 defines information as the usage of triangle patches, *fractional_even_spacing* for the specified spacing between tessellated vertices and a clockwise primitive ordering. These information are also necessary for the primitive generation stage.

```
1  layout(triangles, fractional_even_spacing, cw) in;
2  layout(location = 0) in vec2 tesc_texcoord[];
3
4  layout(binding = 0) uniform UniformBuffer {
5      mat4 model;
6      mat4 view;
7      mat4 proj;
8  };
9
10 layout(binding = 1) uniform sampler2D HeightMap;
11 layout(location = 0) out vec2 tese_texcoord;
```

Code 6.8: Input and output of tessellation evaluation shader.

Since the position of the tessellated vertices are normalized coordiantes, the TES's main function first calculates the real position in world space. Afterwards, texture coordinates for the vertex are calculated and used to extract the corresponding height value of the height map. This is achieved by using the built-in *texture* function, that takes the texture sampler and texture coordinates as input parameter. The extracted height value gets then assigned to the *z* coordinate of the vertex's position. Finally, the world space position is transformed to clip space and assigned to the *gl_Position* variable and passed on together with the calculated texture coordinates to the next stage. After the TES is done, the final mesh of the terrain is generated.

```
1  void main() {
2      vec4 position = gl_in[0].gl_Position * gl_TessCoord.x + gl_in[1].
       gl_Position * gl_TessCoord.y + gl_in[2].gl_Position * gl_TessCoord.z;
3
4      vec2 texcoord = tesc_texcoord[0] * gl_TessCoord.x + tesc_texcoord[1] *
       gl_TessCoord.y + tesc_texcoord[2] * gl_TessCoord.z;
5
6      float height = texture(HeightMap, texcoord).x;
7      position.z += height;
8
9      gl_Position =  proj * view * position;
10     tese_texcoord = texcoord;
11 }
```

Code 6.9: Main function of tessellation evaluation shader.

7 Results and Benchmarks

This chapter will show multiple benchmark results of the introduced terrain renderer. The benchmarks were made of different scenarios, that include different terrain structures, view points and view directions. Each of the scenarios was carried out four times, in order to have representative results. The goal of the benchmarks is to see how the renderer behaves in different situations and to have comparative values for future works. For the values of the benchmark that were taken into account are the frames per second, CPU load, GPU load, memory usage and memory throughput, while each benchmark is recorded over 60 seconds of rendering. For the terrain data a DEM and orthophoto data set of the the region Bad Tölz-Wolfratshausen in Germany was used.

7.1 Embedded Device

As embedded device for the benchmarks was intended to use a Jacinto TDA4VH-Q1 from Texas Instruments. However, when trying to run the application on that device it ended up crashing. This was due to the usage of the tessellation shader stage in the rendering pipeline. The usage of a tessellation shader is a feature that has to be supported by the hardware. Unfortunately, it turned out that the TDA4VH-Q1 board does not support hardware tessellation for Vulkan. Therefore, for testing the terrain renderer and gathering benchmark results it has been fallen back to a Nvidia Jetson TX2.

7.2 General Benchmark Settings

Every scenario uses a different combination of values for the view point position, view direction and terrain area. But, there are factors that are set to the same value for all benchmark scenarios. The maximum view distance is set to *4000.0*, that leads to a chunkmap with a count of 81 chunks. The tessellation factor is set to *1.0* and the images are rendered in a resolution of *1920 x 1080* pixels. The position of the the view point is updated in intervals of *25* ms with a displacement of *2.1* in world space coordinates. This way of updating the position can be interpreted with a theoretical view point movement speed of *288* km/h. Additionally, a field of view angle of *45°* is used. Furthermore, the frequency of the CPU and GPU was set to a static value, so that the load for both can be analyzed easier. For the CPU a frequency of *2.0352* GHz was used, while the GPU's frequency was set to *1.122* GHz.

7.3 Flat Terrain

The first two scenarios use the same terrain area and trajectory for the benchmark recording. For those two a flat terrain area was used. A flat area in this case is defined as terrain with hardly any elevation differences. The trajectory starts from the position [47.7506, 11.46136] in geographical coordinates and moves in a straight line up north to [47.79375, 11.46136]. The trajectory can be seen in figure 7.1, along with the terrain height differences.

Figure 7.1: Shows elevation differences of the used terrain and the trajecty (red line).

7.3.1 Close View

For the first flat terrain scenario, the view point is set close to the terrain surface at a constant height of *670.0* m relative to the sea level. The view direction points to the north and uses a view angle of *63.43°*. Figure 7.2 shows an rendered image from the mentioned scenario and illustrates the the view direction with its view angle.

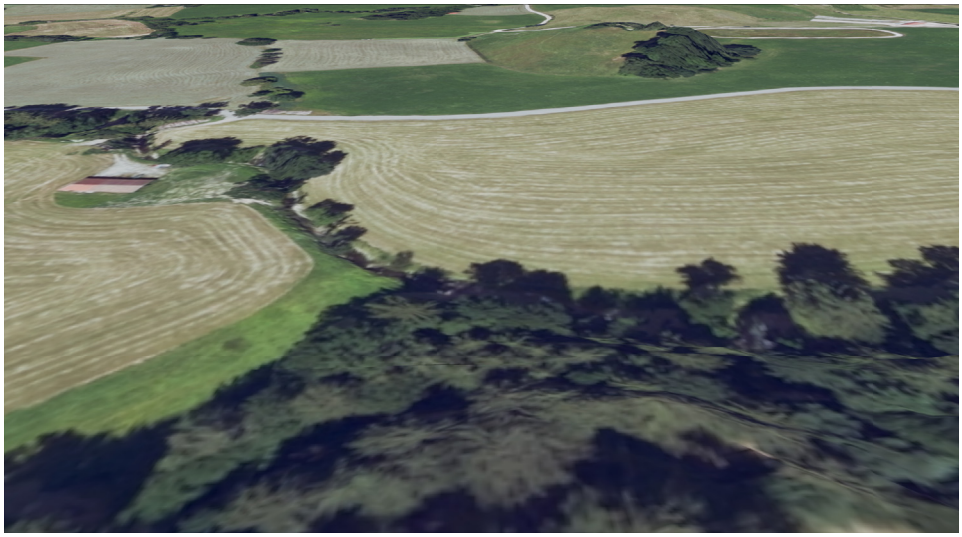


Figure 7.2: Rendered image of the scenario with a close view to the terrain.

7.3.1.1 Frames per Second

As it can be seen in figure 7.3 the frames per second (FPS) are over all four rounds almost equal to each other. The drastic drops are attributed to the chunkmap update process, when it pauses the render process to update the newly loaded and created flat models, texture images and height maps.

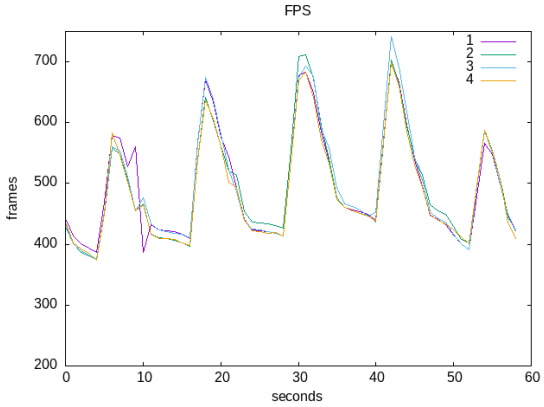


Figure 7.3: Shows FPS for the first scenario.

7.3.1.2 CPU and GPU Load

Both graphs of figure 7.4 show an almost identical behavior of the GPU and CPU load over all four runs. When analyzing both graphs a counterplay between the CPU and GPU can be identified. Whenever the CPU has peaks in its load, the GPU's load drops at the same time. The drastic peaks of the CPU and drops of the GPU are again due to the update process, when the CPU blocks the GPU for updating the chunkmap.

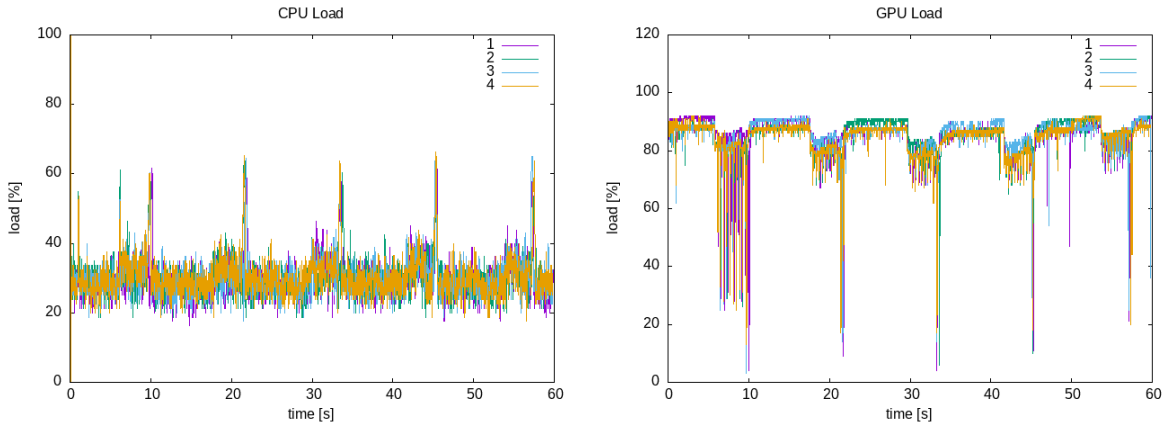


Figure 7.4: Shows CPU and GPU Load of the first scenario.

7.3.1.3 Memory Usage and Throughput

When it comes to the memory usage a constant history with a small increase in the first 10 seconds can be identified. However, this graph does not correspond to the expected behavior. Since the updating process loads new chunks and buffers them before updating the chunkmap, it was expected to be a graph with small peaks, that get back to the same level of memory usage as it was before of the updating process, as the the buffers are freed and the amount of data in the chunkmap does not change. Therefore, the graph of the memory usage does not match with the expectations and its behavior is hard to interpret.

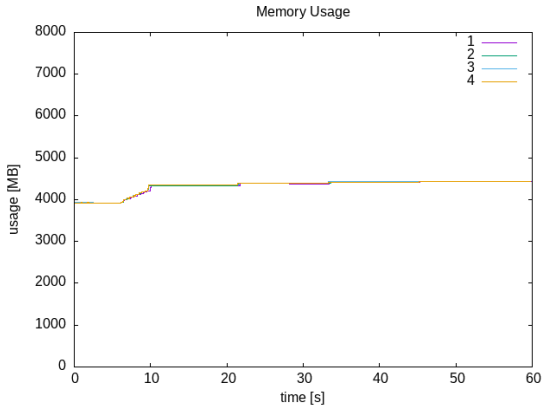


Figure 7.5: Shows memory usage of the first scenario.

In contrast to the memory usage, the graph of the memory throughput matches the expected behavior. The throughput increases whenever new data is loaded by the chunkmap updating process and decreases when it is done.

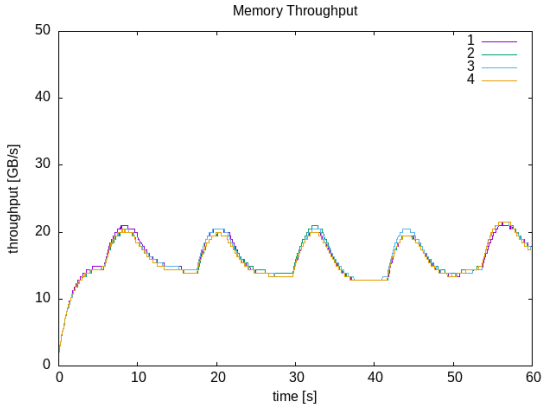


Figure 7.6: Shows memory throughput of the first scenario.

7.3.2 Distant View

For the second flat scenario, the position of the view point is set to a higher position compared to the first scenario, with a height of *1100.0* m relative to the sea level. The view direction points again up north, but with a view angle of *45°*. Figure 7.7 shows an rendered image from the second scenario and illustrates the view direction with its view angle.

Figure 7.7: Rendered image of the second scenario with a distant view to the terrain.

7.3.2.1 Frames per Second

Compared to the FPS of the first scenario, the average FPS are lower. A reason for these results is a larger amount of chunks, that have to be tessellated and used for rendering an image. Additionally, it can be seen, that there are no peaks as there was in the first one. The FPS are constant on an equal level between the loading phases, what might be due to a more constant amount of chunks that are used in the rendering phase.

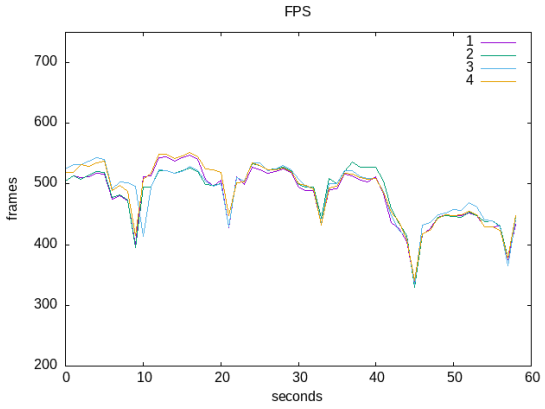


Figure 7.8: Shows FPS of the second scenario.

7.3.2.2 CPU and GPU Load

The graphs of the second scenario's CPU and GPU load match with the results of the first scenario. Therefore, no further analysis is necessary.

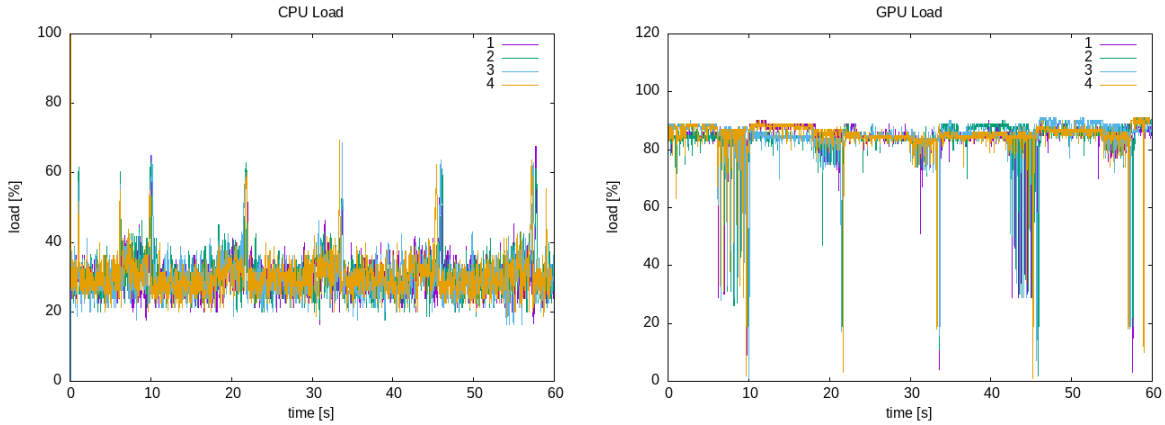


Figure 7.9: Shows CPU and GPU load of the second scenario.

7.3.2.3 Memory Usage and Throughput

The memory usage, again, is not as expected. However, in this case the graph illustrates a different behavior compared to the first time. Instead of being a straight line, a small increase of the memory usage can be seen, when new data is loaded. But, not as expected the memory usage stays the same until the next data is loaded, instead of decreasing. Furthermore, the drop between 30 and 40, that occurs at all four runs, can also not be explained.

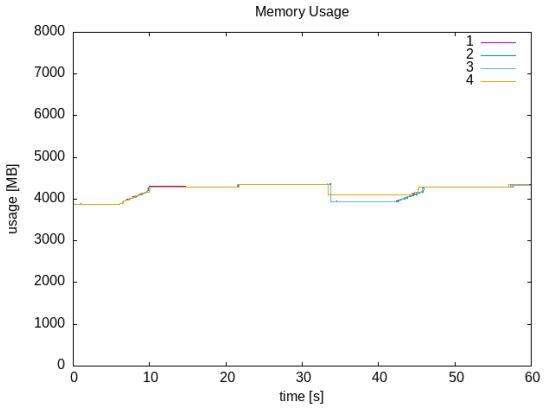


Figure 7.10: Shows memory usage of the second scenario.

The memory throughput also shows a different behavior compared to the first scenario. The maximum memory throughput is higher and it has no significant drops between the chunkmap update phases. However, the basic expected pattern is still existing, but not as distinct as it was in the first scenario.

There is no reasonable explanation for this graph, since the amount of data, that is being loaded by the update processes, stays the same between both scenarios.

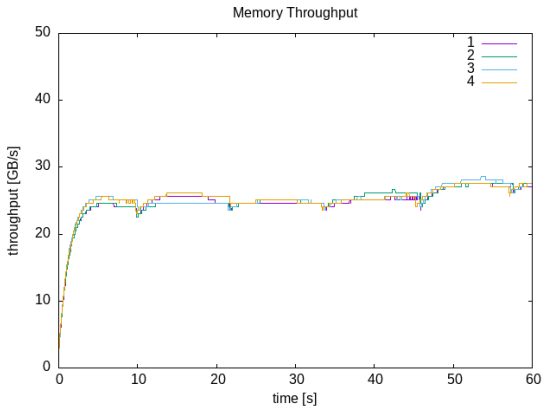


Figure 7.11: Shows memory throughput of the second scenario.

7.4 Rough Terrain

For the last two scenarios a different terrain area and trajectory is used. The terrain area represents a rough terrain. The definition of a rough terrain in this case is an area with many and huge elevation differences. The trajectory starts from the position [47.57249, 11.37322] in geographical coordinates and moves in a straight line to the east to [47.57249, 11.43697]. The trajectory can be seen in figure 7.12, along with the terrain height differences.

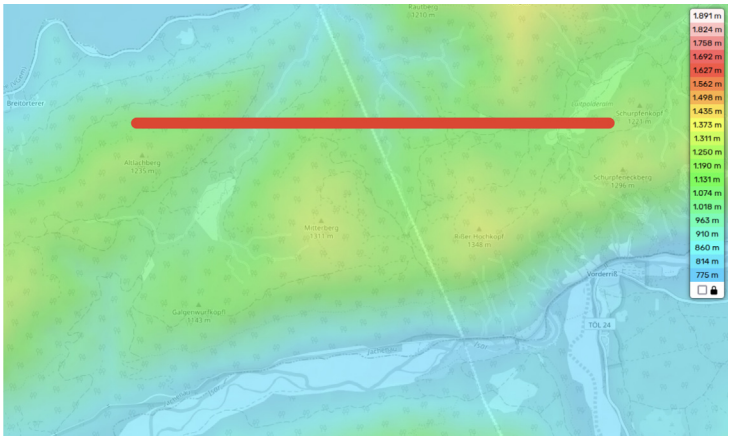


Figure 7.12: Shows elevation differences of the used rough terrain and the trajectory (red line).

7.4.1 Far View

For the third scenario, the view point position is set to *2000.0* m relative to the sea level, while the view direction points to the east, and uses a view angle of *45°*. Figure 7.13 shows a rendered image from the third scenario and illustrates the view direction with its angle.

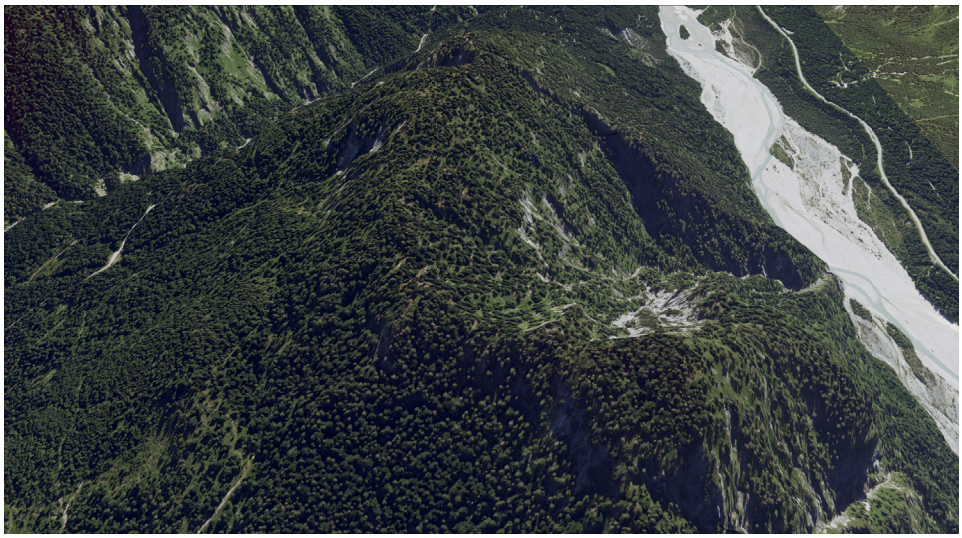


Figure 7.13: Rendered image of the third scenario.

7.4.1.1 Frames per Second

The expected pattern, as it occurred in the first two scenarios, can be seen here as well. However, after the first half of each run, the FPS drop drastically and stay on that lower level. A first thought was, that this behavior is due to the count of chunks, that are considered for rendering. But when taking a look at the right graph of figure 7.14, that shows the number of average chunks per second used for rendering over the course of time, no correlation can be identified.

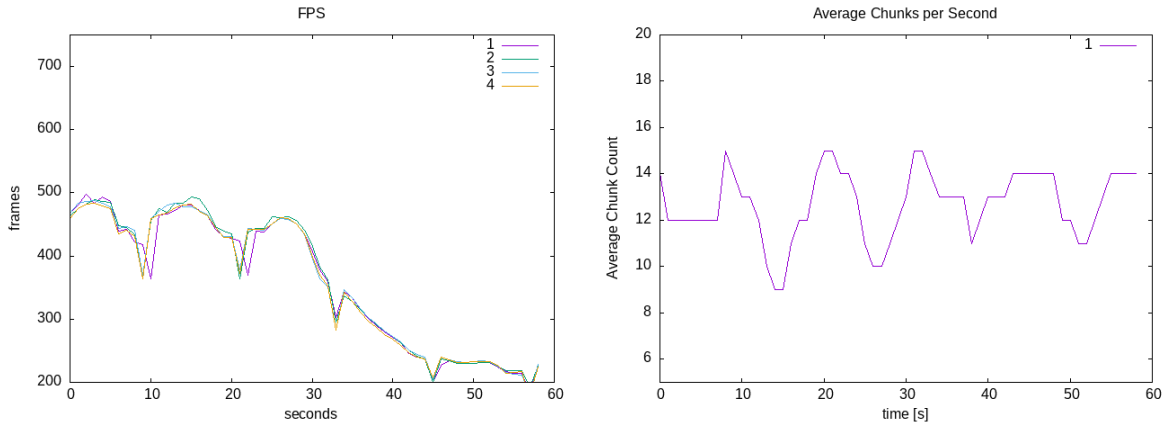


Figure 7.14: Shows FPS and average chunks per second used for rendering for the third scenario.

A thought that might be plausible, is that the area that is rendered when the FPS drops occur consists of more front-facing triangles. During the fragment shader stage, back-face culling is performed. When the area contains a larger amount of front-facing triangles the back-face culling process culls a less amount of triangles and the following stages in the render pipeline have to deal with more triangles.

7.4.1.2 CPU and GPU Load

The graphs of the third scenario's CPU and GPU load match with the results of the previous scenarios. Therefore, no further analysis is necessary.

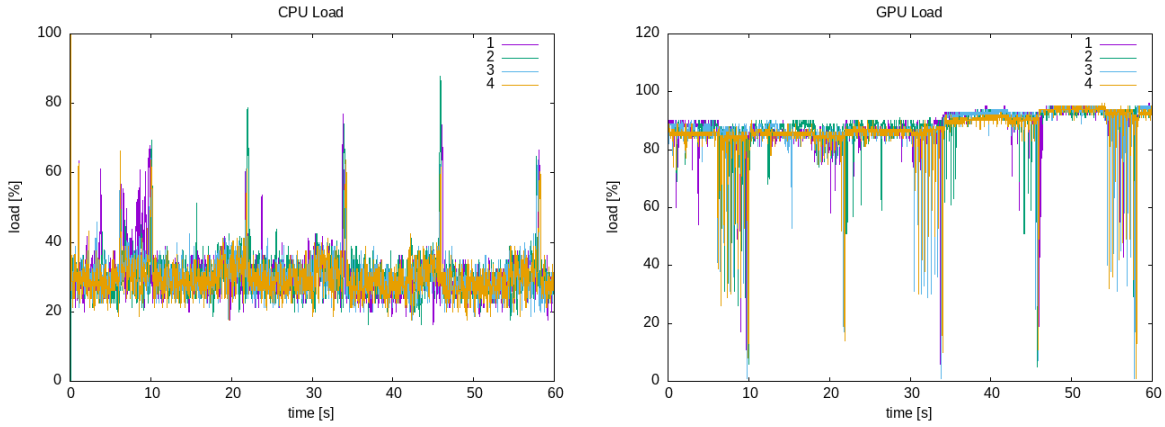


Figure 7.15: Shows CPU and GPU load of the third scenario.

7.4.1.3 Memory Usage and Throughput

The third scenario's memory usage and throughput results are similar to the results of the second scenario, thus no analyzing is needed.

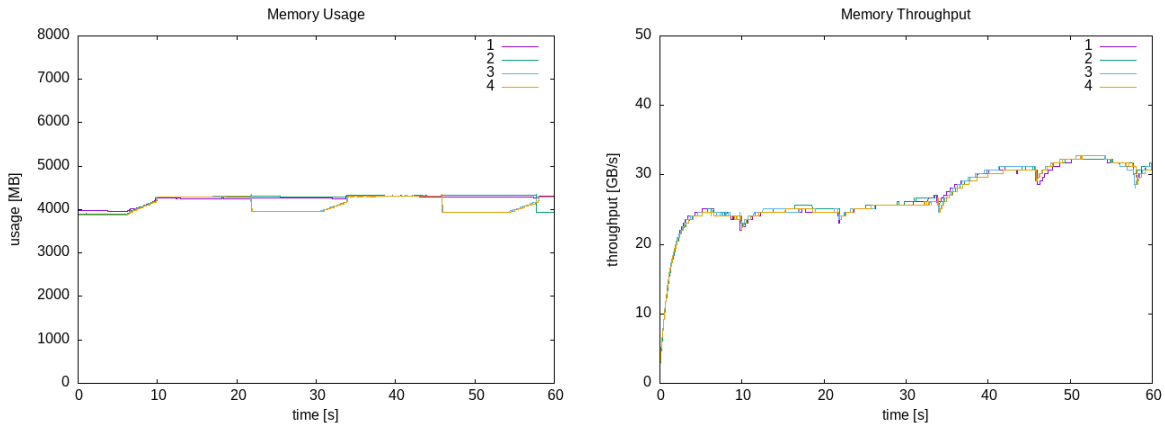


Figure 7.16: Shows memory usage and memory throughput of the third scenario.

7.4.2 Steep View

The fourth scenario uses the same view point height and view direction pointing as the third scenario. Only the the angle of the view direction is changed to a steeper one of approximately *20°*. Figure 7.17 shows a rendered image from the fourth scenario and illustrates the view direction with its angle. The results of the CPU and GPU load and the memory usage and throughput are similar to previous mentioned scenarios, and are therefore just listed for completeness without further descriptions.

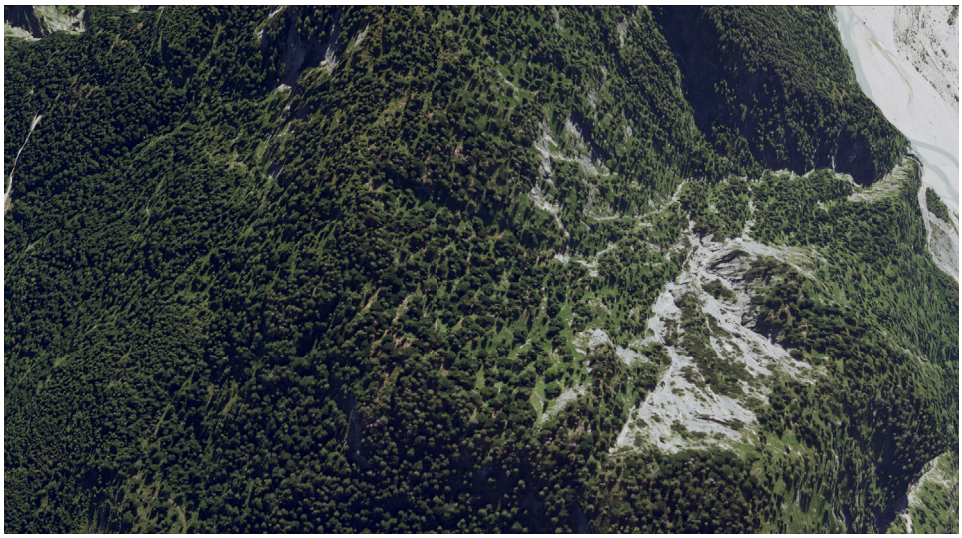


Figure 7.17: Rendered image of the fourth scenario.

7.4.2.1 Frames per Second

The average FPS are higher compared to the third scenario, due to a smaller amount of chunks, that are necessary for rendering an image, as a result of the steeper view angle. But again, as in the previous scenario a significant drop of the FPS is noticeable, but occurs a bit later in the last quarter. This drop could be argued as in the explanation from the third scenario.

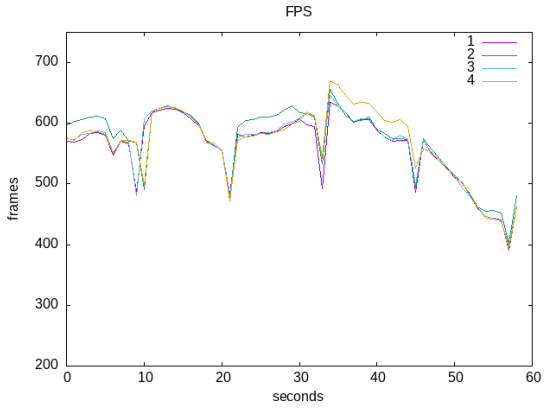


Figure 7.18: Shows FPS of the fourth scenario.

7.4.2.2 CPU and GPU Load

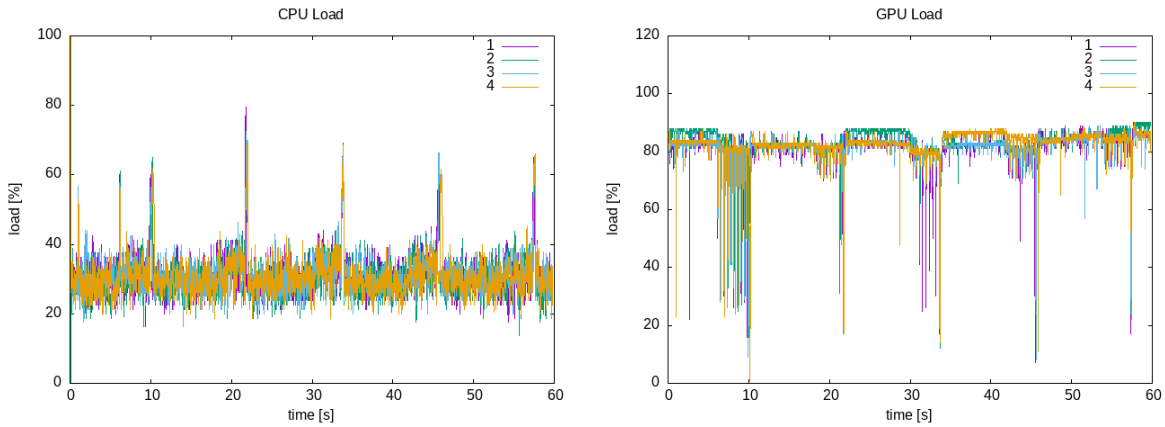


Figure 7.19: Shows CPU and GPU load of the fourth scenario.

7.4.2.3 Memory Usage and Throughput

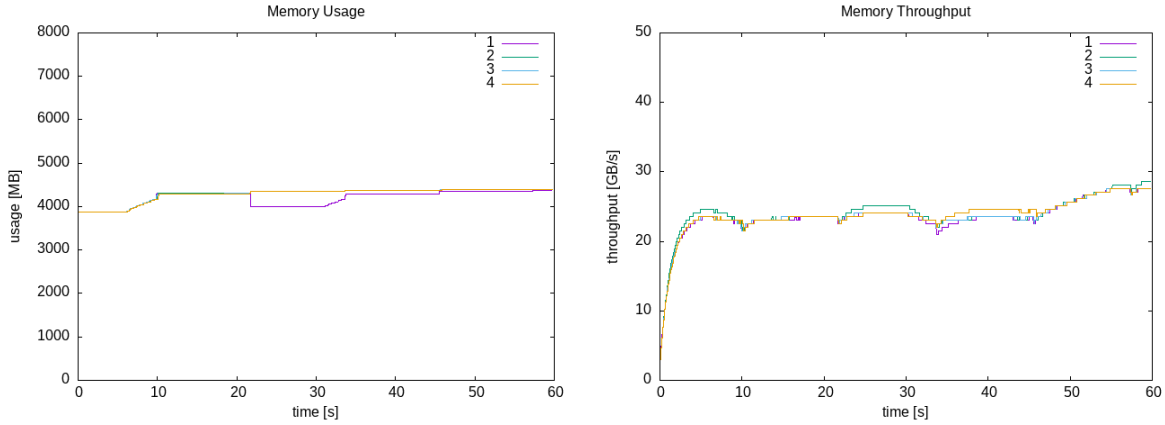


Figure 7.20: Shows memory usage and memory throughput of the fourth scenario.

7.5 Conclusion

The introduced terrain renderer is capable of rendering terrain with a high framerate. As in the benchmark results can be seen, the framerate heavily depends on the distance between the view point and terrain surface and the view angle. Furthermore, terrain areas consisting of a larger amount of front-facing triangles seem to be a significant influence on the FPS. But in most of the presented scenarios, a framerate between *400* and *500* FPS was achieved, while the lowest FPS never dropped under *190* FPS. Therefore, the requirement of one frame every *20* ms can be fulfilled by this thesis's terrain renderer.

8 Summary and Future Work

8.1 Summary

The main target of this thesis was to develop a terrain renderer for real world terrain with following requirements:

- usage of Vulkan as graphics API
- real-world height data and images are organized in multiple tiles of 1km x 1km
- executable on embedded devices with at least one frame every *20*ms
- off-screen rendering

In order to achieve this goal, a basic knowledge of terrain rendering was necessary, that was described and explained in the second chapter.

Since Vulkan is used as the graphics API, it was introduced, along with its basics, as the rendering pipeline and concepts of rendering images, in the third chapter.

To find a suitable rendering method, several different techniques were introduced and analyzed, what resulted in the choice of a hardware tessellation approach.

The final method was then discussed and presented in detail, while important parts of the implementation were analyzed closer by taking a look at their code snippets.

By recording benchmarks of different scenarios, the performance of the terrain renderer was analyzed, along with the behavior of the embedded system's hardware components, as CPU, GPU and memory. As conclusion, it can be said, that the developed terrain renderer is capable of delivering a high framerate, while certain cases of terrain geometry seem to challenge it a bit more and ends up with a drop of the FPS.

8.2 Future Work

When it comes to possible future work regarding the terrain renderer, there are a few further improvements that can be implemented and tested. At first, it can be tried to take advantage of the embedded system's unified memory. So far, images and buffers are transferred between host-visible and device-local memory, when using them on the GPU. The unified memory of embedded devices usually does not differ between those two types of memory. Therefore, it can be tried, to get rid of the memory transfers to different memory types, what might result in a performance gain.
Furthermore, a texture compression method can implemented to reduce the amount memory occupied

by the texture images.
Lastly, benchmarks of an OpenGL terrain renderer of the same scenarios could be recorded, to find out, whether there are performance differences between both graphics APIs.

Acronyms

API application programming interface. 5, 12

CLOD Continuous Level of Detail. 8, 23

DEM digital elevation model. 11, 29, 43

DLOD Discrete Level of Detail. 7

DOP digital orthophoto. 11

DSM digital surface model. 11

FPS Frames per Second. 45, 47, 50, 52–54

GLSL OpenGL Shading Language. 18

LOD Level of Detail. 8, 30, 34

MLOD Multi-resolutional Level of Detail. 8

NDC Normal Device Coordinate. 3, 4

PGM Persistent Grid Mapping. 26

ROAM Real-time Optimally Adapting Meshes. 23, 30

RTIN right-triangulated irregular networks. 5, 6, 23

TCS Tessellation Control Shader. 14, 34, 40–42

TES Tessellation Evaluation Shader. 16, 34, 40, 42

TIN triangulated irregular network. 5, 6

Bibliography

[1] David Luebke et al. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers, 2003. ISBN: 1-55860-838-9.

[2] Mike Bailey. "Introduction to the Vulkan® computer graphics API". In: *SIGGRAPH Asia 2019 Courses*. SA '19. Brisbane, Queensland, Australia: Association for Computing Machinery, 2019. ISBN: 9781450369411. DOI: 10.1145/3355047.3359405. URL: https://doi-org.thi.idm.oclc.org/10.1145/3355047.3359405.

[3] Neeraj Bhargava, Ritu Bhargava, and Prakash Singh Tanwar. "Triangulated Irregular Networkt Model from Mass Points". In: *International Journal of Advanced Computer Research* (2013). URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=7c31254129c72b226d51f2bd6a91a8eaa6840277.

[4] Victor Blanco. *Descriptor Sets*. (visited on 01/30/2024). 2020. URL: https://vkguide.dev/docs/chapter-4/descriptors/.

[5] Victor Blanco. *Executing Vulkan Commands*. (visited on 01/31/2024). 2020. URL: https://vkguide.dev/docs/chapter-1/vulkan_command_flow/.

[6] Victor Blanco. *Vulkan Images*. (visited on 01/30/2024). 2020. URL: https://vkguide.dev/docs/chapter-5/vulkan_images/.

[7] Jonas Bösch, Prashant Goswami, and Renato Pajarola. "RASTeR, Simple and Efficient Terrain Rendering on the GPU". In: *EUROGRAPHICS 2009* (2009). URL: https://maverick.inria.fr/~Prashant.Goswami/Research/Papers/RASTeR_EG09.pdf.

[8] *Core Language (GLSL)*. (visited on 01/27/2024). 2/14/2024. URL: https://www.khronos.org/opengl/wiki/Fragment_Shader.

[9] Nvidia Corporation. *Engaging the Voyage to Vulkan*. (visited on 01/31/2024). URL: https://developer.nvidia.com/engaging-voyage-vulkan.

[10] Nvidia Corporation. *Transitioning from OpenGL to Vulkan*. (visited on 01/20/2024). URL: https://developer.nvidia.com/transitioning-opengl-vulkan.

[11] Nvidia Corporation. *Vulkan Support on Jetson Linux*. (visited on 01/20/2024). URL: https://developer.nvidia.com/embedded/vulkan.

[12] Dilip Kumar Dalei, N Venkataramanan, and Narayan Panigrahi. "A Review of LOD based Techniques for Real-time Terrain Rendering". In: *2022 IEEE 6th Conference on Information and Communication Technology (CICT)*. 2022, pp. 1–6. DOI: 10.1109/CICT56698.2022.9997973.

[13] *Digital elevation model*. (visited on 01/04/2024). URL: https://en.wikipedia.org/wiki/Digital_elevation_model.

[14] M. Duchaineau et al. "ROAMing terrain: Real-time Optimally Adapting Meshes". In: *Proceedings. Visualization '97 (Cat. No. 97CB36155)*. 1997, pp. 81–88. DOI: 10.1109/VISUAL.1997.663860.

[15] *Face Culling*. (visited on 01/12/2024). 10/18/2019. URL: https://www.khronos.org/opengl/wiki/Face_Culling.

[16] *Fragment Shader*. (visited on 01/27/2024). 11/25/2020. URL: https://www.khronos.org/opengl/wiki/Fragment_Shader.

[17] Michael Garland and Paul S. Heckbert. *Fast Polygonal Approximation of Terrains and Height Fields*. Tech. rep. Carnegie Mellon University, 1995. URL: https://www.cs.cmu.edu/~garland/scape/scape.pdf.

[18] *Geometry Shader*. (visited on 01/27/2024). 11/23/2022. URL: https://www.khronos.org/opengl/wiki/Geometry_Shader.

[19] GISGeography. *DEM, DSM DTM: Elevation Models in GIS*. (visited on 01/14/2024). 1/06/2024. URL: https://gisgeography.com/dem-dsm-dtm-differences/.

[20] Chua Hock-Chuan. *3D Graphics with OpenGL*. (visited on 01/23/2024). 2012. URL: https://medium.com/@rakadian/vertex-dc7b0eeeb769.

[21] Goanghun Kim and Nakhoon Baek. "A Height-Map Based Terrain Rendering with Tessellation Hardware". In: *2014 International Conference on IT Convergence and Security (ICITCS)*. 2014, pp. 1–4. DOI: 10.1109/ICITCS.2014.7021710.

[22] Goanghun Kim and Nakhoon Baek. "A Height-Map Based Terrain Rendering with Tessellation Hardware". In: *2014 International Conference on IT Convergence and Security (ICITCS)*. 2014, pp. 1–4. DOI: 10.1109/ICITCS.2014.7021710.

[23] Sudobh Kumar et al. *Hierarchical Back-Face Culling*. Tech. rep. Universitry of North Carolina. URL: https://wwwx.cs.unc.edu/~geom/papers/documents/technicalreports/tr96014.pdf.

[24] Pawel Lapinski. *API without Sectres: Introduction to Vulkan* Part 5*. (visited on 02/02/2024). 8/12/2016. URL: https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-part-5.html.

[25] Pawel Lapinski. *Vulkan Cookbook*. Packt Publishing, 2017. ISBN: 1786468158.

[26] Raphaël Lerbour. *Adaptive streaming and rendering of large terrains*. Tech. rep. Human-Computer Interaction, 2009. URL: https://theses.hal.science/tel-00461667/document.

[27] Peter Lindstrom and Valerio Pascucci. "Visualization of large terrains made easy". In: *Proceedings of the Conference on Visualization '01*. VIS '01. San Diego, California: IEEE Computer Society, 2001, 363–371. ISBN: 078037200X.

[28] 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*. SIGGRAPH '96.
New York, NY, USA: Association for Computing Machinery, 1996, 109–118.
ISBN: 0897917464. DOI: 10.1145/237170.237217.
URL: https://doi.org/10.1145/237170.237217.

[29] Yotam Livny et al. "A GPU persistent grid mapping for terrain rendering".
In: *Visual Comput* (2007). URL: https://www.cs.bgu.ac.il/~elsana/publications/pdf/PersistentGridMapping.pdf.

[30] Frank Losasso and Hugues Hoppe.
"Geometry clipmaps: terrain rendering using nested regular grids".
In: *ACM Trans. Graph.* 23.3 (2004), 769–776. ISSN: 0730-0301.
DOI: 10.1145/1015706.1015799.
URL: https://doi-org.thi.idm.oclc.org/10.1145/1015706.1015799.

[31] LunarG. *Create a Uniform Buffer*. (visited on 01/30/2024). 2016.
URL: https://vulkan.lunarg.com/doc/view/latest/windows/tutorial/html/07-init_uniform_buffer.html.

[32] LunarG. *Create the Framebuffers*. (visited on 01/30/2024). 2016.
URL: https://vulkan.lunarg.com/doc/view/latest/windows/tutorial/html/12-init_frame_buffers.html.

[33] Mapasyst. *What is an orthophoto?* (visited on 01/14/2024). 8/21/2019.
URL: https://mapasyst.extension.org/what-is-an-orthophoto/.

[34] *OpenGL ES*. (visited on 01/20/2024). 5/21/2020.
URL: https://www.khronos.org/opengl/wiki/OpenGL_ES.

[35] Bruno Opsenica. *Frustum Culling*. (visited on 01/12/2024). 12/24/2020.
URL: https://bruop.github.io/frustum_culling/.

[36] Alexander Overvoorde. *Fixed functions*. (visited on 01/26/2024). URL: https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Fixed_functions.

[37] Alexander Overvoorde. *Graphics pipeline basics*. https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction (visited on 02/02/2024).

[38] Alexander Overvoorde. *Index Buffer*. (visited on 02/02/2024).
URL: https://vulkan-tutorial.com/Vertex_buffers/Index_buffer.

[39] Alexander Overvoorde. *Rendering and presentation*. (visited on 02/02/2024).
URL: https://vulkan-tutorial.com/Drawing_a_triangle/Drawing/Rendering_and_presentation.

[40] PacktPublishing. *3D-Graphics-Rendering-Cookbook*. (visited on 02/10/2024).
URL: https://github.com/PacktPublishing/3D-Graphics-Rendering-Cookbook/tree/master/shared.

[41] Jeremy Gebben Raphael Mun John Zulauf and Jan-Harald Fredriksen.
Understanding Vulkan Synchronization. (visited on 02/02/2024). 3/24/2021.
URL: https://www.khronos.org/blog/understanding-vulkan-synchronization.

[42] *Rasterization*. (visited on 01/27/2024).
URL: https://docs.vulkan.org/spec/latest/chapters/primsrast.html.

[43] *Rendering Pipeline Overview*. (visited on 01/20/2024). 11/07/2022.
URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview.

[44] *Resource Descriptors*. (visited on 01/30/2024).
URL: https://docs.vulkan.org/spec/latest/chapters/descriptorsets.html.

[45] *Samplers*. (visited on 01/30/2024).
URL: https://docs.vulkan.org/spec/latest/chapters/samplers.html.

[46] Parminder Singh. *Learning Vulkan*. Packt Publishing, 2016. ISBN: 1786469804.

[47] *Tessellation*. (visited on 01/27/2024).
URL: https://docs.vulkan.org/spec/latest/chapters/tessellation.html.

[48] *Tessellation*. (visited on 01/20/2024). 10/11/2020.
URL: https://www.khronos.org/opengl/wiki/Tessellation.

[49] Thatcher Ulrich. *Rendering Massive Terrains using Chunked Level of Detail Control*.
Tech. rep. Oddworld Inhabitants, 2002.
URL: https://www.classes.cs.uchicago.edu/archive/2015/fall/23700-1/final-project/chunked-lod.pdf.

[50] Bayerische Vermessungsverwaltung. *Kostenfreie Geodaten (OpdenData)*.
(visited on 01/10/2024). URL: https://geodaten.bayern.de/opengeodata/index.html.

[51] *Visibility and Occlusion Culling*. (visited on 01/12/2024). URL: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/VisibilityCulling/.

[52] *VkImageView(3) Manual Page*. (visited on 01/30/2024). 2/16/2024.
URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkImageView.html.

[53] Joey de Vries. *Coordinate Systems*. (visited on 02/10/2024).
URL: https://learnopengl.com/Getting-started/Coordinate-Systems.

[54] University of Waterloo. *Backface Culling*. (visited on 01/12/2024).
URL: http://medialab.di.unipi.it/web/IUM/Waterloo/node66.html.

[55] Sascha Willems. *Vulkan*. (visited on 12/02/2023).
URL: https://github.com/SaschaWillems/Vulkan.

[56] J. Willmott et al.
"Rendering of large and complex urban environments for real time heritage reconstructions".
In: *Proceedings of the 2001 Conference on Virtual Reality, Archeology, and Cultural Heritage*.
VAST '01. Glyfada, Greece: Association for Computing Machinery, 2001, 111–120.
ISBN: 1581134479. DOI: 10.1145/584993.585012.
URL: https://doi-org.thi.idm.oclc.org/10.1145/584993.585012.