LOD terrain rendering of SRTM data using mesh shaders in Vulkan

(Renderowanie terenów używając danych SRTM
i mesh shaders w Vulkanie z optymalizacją LOD)

Cezary Czubała

Praca inżynierska

Promotor: dr Andrzej Łukaszewski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

27 stycznia 2025

Abstract

With the release of AMD's Radeon RX 6000 series and NVIDIA's RTX 20 series, a new technology called mesh shaders was introduced and given full support. It serves as an alternative to the traditional graphics rendering pipeline and despite the mostly positive reception, the technology is still rarely used in commercial software. In this thesis, I will give an overview of the technology, implement a terrain renderer with LOD (Level of Detail) optimizations, and compare the development process to that of a traditional pipeline.

Wraz z wydaniem kart serii Radeon RX 6000 od AMD i RTX 20 od NVIDIA, wsparcie uzyskała nowa technologia zwana mesh shaders. Proponuje ona alternatywę dla tradycyjnego potoku graficznego i mimo pozytywnych opinii, mesh shadery rzadko spotyka się w komercyjnych programach. W tej pracy opiszę na czym polega ta technologia, zaimplementuję program do renderowania terenu z optymalizacjami LOD (Level of Detail) i porównam proces tworzenia aplikacji z tradycyjnym potokiem.

Contents

Chapter 1

Introduction

The traditional graphics pipeline at its most basic level is structured in the following way: a fixed input assembly stage receives vertex and primitive input data, passes it onto the vertex shader, which further is rasterized and processed per pixel in the fragment shader. This approach is too basic for modern graphic engines, and so additional programmable shader stages were added: tessellation and geometry [1]. Although tessellation shaders find use in most engines, geometry shaders are seldom found due to their often poor performance.

A new pipeline was proposed that makes use of mesh shaders to reduce the number of programmable stages and give more flexibility in terms of mesh generation and input data. In order to explore potential uses of this technology, an implementation of a terrain rendering application will be presented and the differences between the two pipelines will be discussed, as well as several setbacks developers may face.

Chapter 2

Mesh shaders

2.1 Overview

The first mention of mesh shaders can be traced back to late 2018 by Christoph Kubisch[2]. The basis of designing mesh shaders was to improve rendering high-resolution meshes required for outside environments or CAD meshes. They also introduced a structure similar to compute shaders allowing for a design based on work groups. Mesh shaders aren't limited by an input assembly — any data our shaders require has to be passed through buffer objects such as SSBOs or UBOs. An example of mesh shader usage in the commercial market is the game "Alan Wake 2" developed by Remedy Entertainment.

An example implementation of mesh rendering using mesh shaders in Vulkan was provided by NVIDIA in the `gl_vk_meshlet_cadscene` project[3].

2.1.1 Meshlets

Meshlets are a key element of the mesh shader workflow — they are defined as a mesh with an upper limit of primitives and vertices [4] — ideally, they maximize the ratio of indexed primitives to unique vertices. They can be visualized as small chunks or subdivisions of a given mesh (figure 2.1). The mesh can be split either manually or via automatic pre-processing into such meshlets, which are then used in mesh shader invocations as input data to generate the final geometry. The optimal sizes for such meshlets depend on the vendor preferences, which can be queried at runtime.

2.1.2 Mesh shader pipeline structure

The new pipeline (figure 2.2) proposed with the mesh shader technology is made up of an optional task shader and a compulsory mesh shader stage. The task shader

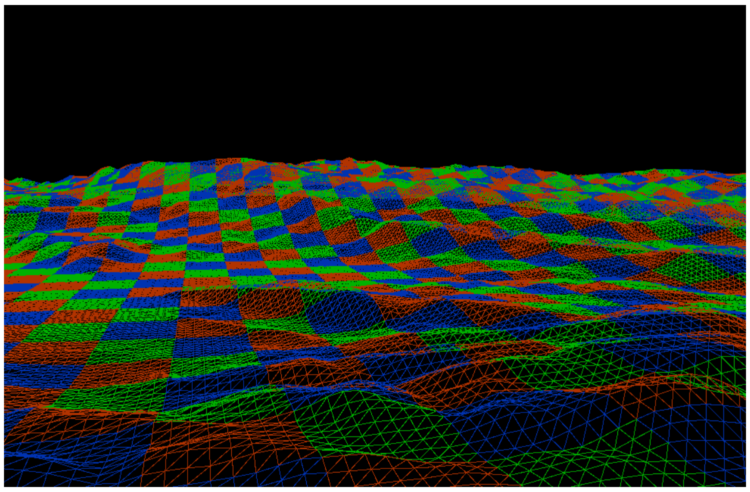


Figure 2.1: An example visualization of meshlets

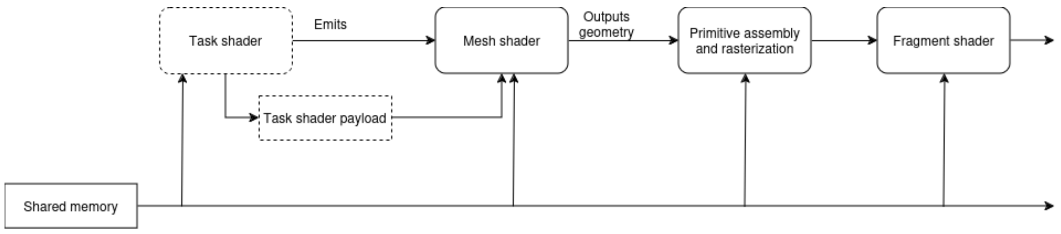


Figure 2.2: The mesh shader pipeline

dispatches a specified number of mesh shader work groups and may pass to them a small amount of data in the form of a payload[5]. Due to the lack of an input assembly, the input mesh data may be passed through buffers such as shader storage buffer objects. The mesh shader further generates the geometry we want rasterize.

2.1.3 Mesh shader program example

Listing 2.1 presents an example mesh shader that renders pre-calculated meshlets without the use of task shaders. Lines $5-8$ define the limits and what type of primitives the shader will output; line 5 bears resemblance to compute shaders, as we declare the size of mesh shader work groups. Lines $10-25$ declare the input data that will be processed by our shader. Unlike in a traditional vertex shader, the data is passed through SSBOs. The lines $44, 51, 52$ and 56 are worth of notice as they make use of mesh-shader-specific API to output geometry that is then passed onto the rasterizer.

```
#version 460

#extension GL_EXT_mesh_shader : enable

layout (local_size_x=MAX_INVOCATIONS) in;

layout(max_vertices=MAX_OUT_VERT, max_primitives=MAX_OUT_PRIM) out;
layout(triangles) out;

layout(std430, binding = 0) readonly buffer MeshletPrimitivesData {
    uvec3 meshletPrimitivesData[];
};

layout(std430, binding = 1) readonly buffer MeshletVertexData {
    vec3 meshletVertexData[];
};

struct MeshletDescription {
    int vertexDataOffset, vertexCount;
    int primitivesDataOffset, primitiveCount;
};

layout(std140, binding = 2) readonly buffer MeshletDescriptions {
    MeshletDescription meshletDescriptions[];
};

layout( push_constant ) uniform constants {
  mat4 transformationMatrix;
    int  baseMeshletOffset;
} PushConstants;

uint meshletID = gl_GlobalInvocationID.x;

void main() {
    MeshletDescription myMeshlet;
    meshletID = PushConstants.baseMeshletOffset + meshletID;
    myMeshlet = meshletDescriptions.descriptions[meshletID];

    uint vertCount = myMeshlet.vertexCount;
    uint primCount = myMeshlet.primitiveCount;

    for (uint i = 0; i < vertCount; i++) {
        vec3 vertex = meshletVertexData[myMeshlet.vertexDataOffset + i];
        gl_MeshVerticesEXT[i].gl_Position =
            PushConstants.transform_matrix * vec4(vertex, 1.0);
    }

    for (uint i = 0; i < primCount; i++) {
        uvec3 primitive =
                meshletPrimitiveData[myMeshlet.primitiveDataOffset + i];
        gl_PrimitiveTriangleIndicesEXT[i] = primitive;
        gl_MeshPrimitivesEXT[i].gl_PrimitiveID =
            myMeshlet.primitiveDataOffset + i;
    }

    SetMeshOutputsEXT(vertCount, primCount);
}
```

Listing 2.1: Mesh shader code example

2.1.4 Task shaders

An additional shader program can be written and compiled, called the task shader. This stage allows for initial pre-processing of data and dispatching a certain amount of mesh shaders — this allows for early meshlet culling, LOD selection, and other algorithms or optimizations.

2.1.5 Task shader program example

The task shader example in listing 2.2 is simpler than the example mesh shader program (listing 2.1) because its function is also simpler. Lines 14 – 19 present the core feature of a task shader: the task shader payload. In the example code the *meshletID* used in the mesh shader will need to be calculated with the inclusion of both the invocation ID of the mesh shader and the task shader's invocation ID — the reasoning for this is further explained in 2.2. Lastly, line 28 is a declaration of how many mesh shaders should be emitted from this task shader invocation.

```
 1 #version 460
 2 
 3 #extension GL_EXT_mesh_shader : require
 4 
 5 layout (local_size_x=MAX_TASK_INVOCS) in;
 6 
 7 layout( push_constant ) uniform constants
 8 {
 9     uint taskCallMeshletOffset;
10 } PushConstants;
11 
12 uint baseID = gl_WorkGroupID.x;
13 
14 struct Task {
15     uint   baseMeshletIDOffset;
16     vec3   additionalData;
17 };
18 
19 taskPayloadSharedEXT Task OUT;
20 
21 void main() {
22     OUT.baseMeshletIDOffset = PushConstants.taskCallMeshletOffset +
23                               baseID * MAX_MESH_INVOCS;
24     OUT.additionalData = 42;
25 
26     EmitMeshTasksEXT(MAX_MESH_INVOCS, 1, 1);
27 }
```

Listing 2.2: Task shader code example

2.2 Mesh shaders in the Vulkan API

Two extensions add support for mesh shaders in Vulkan: NVIDIA's `VK_NV_mesh_shader` and the cross-vendor `VK_EXT_mesh_shader`. For this project, the `VK_EXT_mesh_shader` extension will be used, although the differences between the two implementations would be minimal. In order to use the extensions, when creating the logical device with `vkCreateDevice` the extension features responsible for the task and mesh shaders in `VkPhysicalDeviceMeshShaderFeaturesEXT` should be set to true[6]. This will allow us to create task and mesh shader stages as well as to make mesh-shader-specific draw calls.

When creating the graphics pipeline, as mentioned in 2.1.2, the user may create a pipeline consisting only of the mesh and fragment shaders, or include all stages, including the task shader stage. The appropriate bits for mesh and task shader stages are `VK_SHADER_STAGE_TASK_BIT_EXT` and `VK_SHADER_STAGE_MESH_BIT_EXT` and they must be used when creating the graphics pipeline.

Finally, when recording the command buffer, the Vulkan API exposes the following draw calls: `vkCmdDrawMeshTasksEXT`, `vkCmdDrawMeshTasksIndirectEXT` and `vkCmdDrawMeshTasksIndirectCountEXT`[7]. These calls work for both variants of the mesh shader pipeline.

Chapter 3

Terrain renderer

In order to showcase an example implementation of a renderer that uses the mesh shader technology, a terrain renderer is implemented using Vulkan and the `VK_EXT_mesh_shader` extension.

3.1 Project setup

The application used to render the SRTM data with mesh shaders was written in C++ and the following tools and dependencies were used:

- Vulkan SDK - for Vulkan usage.
- GLFW - a library for desktop development. It allows us to create a window, process input, display rendered frames etc..
- shaderc - tools required for shader compilation. Especially libshaderc that allows us to compile shaders at runtime.
- Dear ImGui - used for GUI rendering.
- stb_image - used for loading textures.
- GLM - a popular math library.
- CMake - used for compiling the project binaries.

The project was developed on Ubuntu 24.04 and the source code is hosted on GitHub under the following URL: https://github.com/Cez02/LOD-terrain-renderer-thesis[8].

3.1.1 Layout

The project features several key modules implemented as C++ classes. They are responsible for GUI handling, heightmap loading, and Vulkan rendering.

Renderer

The `Renderer` class declared in `tr_renderer.hpp` handles the core rendering logic of the application. The `initVulkan` and `initVKSceneElements` methods are responsible for initializing the renderer and the heightmaps in the scene respectively. When initializing the renderer we need to enable required extensions, enable the mesh shader features, create the swapchains, descriptor pools, command pools, render passes and other key elements needed for rendering a Vulkan scene. The second function that initializes the scene also initializes the heightmaps used during rendering.

The `drawFrame` method is responsible for recording a command buffer, binding the appropriate buffers and making the draw calls for each heightmap. Additionally, the `GUIHandler` is used to draw the current GUI and the rendering statistics are reset for the next frame.

Finally, at the end of the application's runtime, the Vulkan renderer needs to be de-initialized, and all created pools and buffers need to be destroyed. This is done in the `cleanup` method.

Heightmap

The `Heightmap` class declared in `tr_heightmap.hpp` is an instance of a loaded heightmap. The heightmap is created with a path to its respective SRTM file path and later initialized by the Renderer — this initialization involves loading the file and reading its elevation values (which are also laid out in a specific order described in 3.3), reading the file's name as geographic coordinates and creating the SSBOs keeping the heightmap values.

Additionally, during initialization, the heightmap's grid is split into chunks of 9×9 which will serve as our meshlets. These meshlets overlap each other by one unit on each side (figure 3.1) and each will generate 64 quads. This length allows for rendering meshlets in four resolutions: 100%, 50%, 25% and 12.5%.

SRTM files contain 1201×1201 grids for SRTM-3 and 3601×3601 grids for SRTM-1, using 2-byte signed integers — the values represent elevation values sampled at 3 and 1 arc seconds respectively[9]. The terrain renderer will support both SRTM-1 and SRTM-3 file formats. The files are also named according to the geographic coordinates they represent; this is required as the file itself only contains raw elevation data.

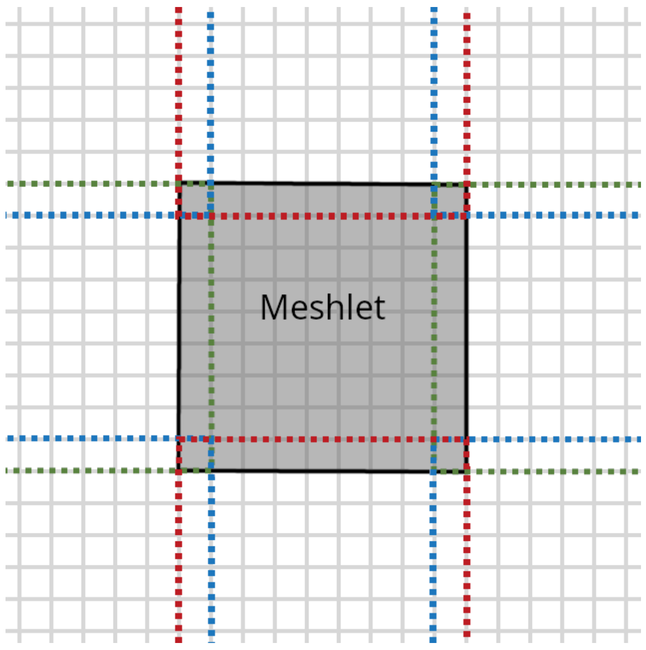


Figure 3.1: An example meshlet built from the heightmap with overlapping meshlet borders visualized. Each cell is one elevation value of the SRTM file.

Each heightmap can also be drawn with the `draw` method. This binds the necessary buffers and creates draw calls that dispatch the vendor-preferred number of task shader invocations.

Shader compiler

Shaders in the terrain renderer program are compiled during initialization at runtime with the use of the shaderc library[10]. This is done in order to pass on compile-time definitions that abide by vendor preferences which can only be queried at runtime when picking the physical device.

The terrain renderer reads the header file `common_template.h` and replaces certain patterns with the required values. Next this new version of `common_template.h` is written to `common.h` which is then included in all shader programs. This allows us to pass the defined meshlet length, workgroup invocation count per each stage and other key properties as preprocessor definitions. Listings 3.1 and 3.2 present an example usage of such pattern replacement, and listing 3.3 contains the exact properties defined in `common_template.h`.

```
#define MAX_PREFERRED_MESH_WORK_GROUP_INVOCATIONS
    $m_MaxPreferredMeshWorkGroupInvocations

struct meshletDescription {
    ...
}
```

Listing 3.1: `common_templates.h` example code fragment, before shader compilation.

```
#define MAX_PREFERRED_MESH_WORK_GROUP_INVOCATIONS 32

struct meshletDescription {
    ...
}
```

Listing 3.2: `common.h` example code fragment, after shader compilation.

```
#define MAX_PREFERRED_MESH_WORK_GROUP_INVOCATIONS
    $m_MaxPreferredMeshWorkGroupInvocations
#define MAX_PREFERRED_TASK_WORK_GROUP_INVOCATIONS
    $m_MaxPreferredTaskWorkGroupInvocations
#define MAX_MESH_OUTPUT_VERTICES $m_MaxMeshOutputVertices
#define MAX_MESH_OUTPUT_PRIMITIVES $m_MaxMeshOutputPrimitives

#define MESHLETS_PER_TASK_INVOCATION $m_MeshletsPerTaskInvocation
#define MESHLETS_PER_MESH_WORKGROUP $m_MeshletsPerMeshWorkGroup

#define MESHLET_LENGTH $meshletLength
```

Listing 3.3: All definitions contained in `common_temlates.h` that will be included during shader compilation.

In order to compile a shader program using shaderc, the original shader source code has to pass through three stages: preprocessing, compilation of GLSL code to SPIR-V code and finally the assembly of the SPIR-V code into a SPIR-V binary file. The shaderc library also exposes several compilation options such as choosing the target SPIR-V or Vulkan version.

During development of the terrain renderer application an error was found causing the SPIR-V version to always be set to 1.0 — a fixed version of the compiled shaderc library is included with the project's source code.

3.2 Usage

The terrain renderer can be launched with the following command-line options:

- `--starting-position <COORDINATES>` - this option sets the player's position to the specified `COORDINATES` which have to follow the SRTM file naming convention. Example usage: `--starting-position N34E014`. By default, the starting position is near `N42E013`

- `--heightmaps-directory <DIRECTORY>` - this option informs the program from which directory the input heightmaps should be read. This directory has to feature only the desired SRTM files. Example usage: `--heightmaps-directory ./heightmaps`. By default, the terrain renderer is launched with the option set to `./heightmaps`.

Upon launch, the terrain renderer will load the heightmaps and once the program is running, the observer can be moved with the following inputs:

- The `WASD` are used for moving the player forward, left, backwards and right respectively.
- The `Spacebar` and `CTRL` keys are used to gain and lose altitude respectively.
- The `E` and `Q` keys are used for rotating the observer right and left respectively.

Additionally, the `Left shift` key may be held while moving in any direction in order to increase the movement speed.

3.3 Rendering

Meshlets and data organization

For this terrain rendering program, the definition of meshlets differs from the traditional as there are no defined meshes and models in the program — the meshlet description (listing 3.4) only serves the purpose of dictating which parts of a given heightmap to render. Although it has been established in 3.1.1 that all meshlets are of 9×9 dimensions, the dimensions of a given meshlet are still specified in its description to allow experimentation and meshlet sizes that are not divisors of the heightmaps size. The `HeightmapDataOffset` field specifies the place of the meshlet elevation values in the heightmap data buffer.

```
struct MeshletDescription {
    uvec2 Offset;
    uvec2 Dimensions;
    uint HeightmapDataOffset;
};
```

Listing 3.4: Meshlet description

Meshlet descriptions and heightmap elevation data are stored in two respective buffers: `meshletDescriptions` and `heightmapData`. The meshlet descriptions are stored in the order they were built; however, the elevation data is stored in chunks corresponding the each meshlet — all elevation values required for a given meshlet are stored in one contiguous chunk row-by-row to improve locality.

One optimization that is possible here is encoding the meshlet into a smaller size, dedicating 2 bytes per `Offset` and `Dimensions` fields to improve bandwidth usage. However, this may not be as beneficial if the decoding method's speed offsets the improved bandwidth usage. Such encoding may be beneficial in traditional, model-based scenarios where more data needs to be stored in the meshlet description.

Task shader

The task shader of the terrain renderer performs several tasks: early meshlet culling, LOD selection, meshlet sorting and mesh shader emission.

The task shader works in work groups made up of 32 invocations, where each work group contains a shared 2D `u16vec2 meshletBuckets[LOD levels][n]` buffer where `n` is the maximum number of meshlets processed per work group. Each task shader invocation iterates over `MESHLETS_PER_TASK_INVOCATION` meshlets, decides whether to cull the meshlet, calculates their LOD level and emplaces the meshlet information (figure 3.3) into the respective `meshletBuckets` shared buffer according to its LOD level in the form of a `meshletInfo` object (figure 3.3).

The information regarding meshlets is passed onto the mesh shader program via the task shader payload. The payload (listing 3.5) contains the data required for final primitive creation in the mesh shader stage.

```
struct TaskPayload {
    u16vec2 meshletClusters[n];
    u16vec2 meshletInfoBuffer[n];

    uint meshShaderProcessingUnitsCount;
    uint meshletInfoBufferCount;

    uint baseMeshletID;
    uint heightmapLength;

    float longitude;
    float latitude;

    vec3 observerPosition;

    uint globalMeshletCount;
};
```

Listing 3.5: The structure used as the task shader payload passed onto the mesh shader stage. n is defined as the maximum number of meshlets processed by the task shader work group.

Once all invocations have processed their meshlets, the first invocation with `gl_InvocationID.x` being equal to 0 performs an algorithm (listing 3.6) that creates meshlet clusters (figure 3.3) out of the prepared meshlet info objects — these units are further used in the mesh shader work groups.

```
declare currentCluster

for LODlevel in 0..3:
    currentCluster = new()

    let maxClusterSize = 1 << (2*LODlevel)

    for meshlet in meshletBuckets[LODlevel]:

        add meshlet to currentCluster

        if currentCluster contains maxClusterSize meshlets:
            add currentCluster to taskShaderPayload.meshletClusters
            currentCluster = new()

    if currentCluster not empty and not added:
        add currentCluster to taskShaderPayload.meshletClusters
```

Listing 3.6: Meshlet cluster building algorithm

The algorithm from listing 3.6 creates meshlet clusters that are responsible for drawing sets of meshlets that share the same LOD level. The number of these meshlets in each processing unit is dictated by the LOD level of these meshlets; the reason for this is explained in 3.3. After the algorithm is finished, the invocation emits as many mesh shader work groups as there are meshlet clusters.

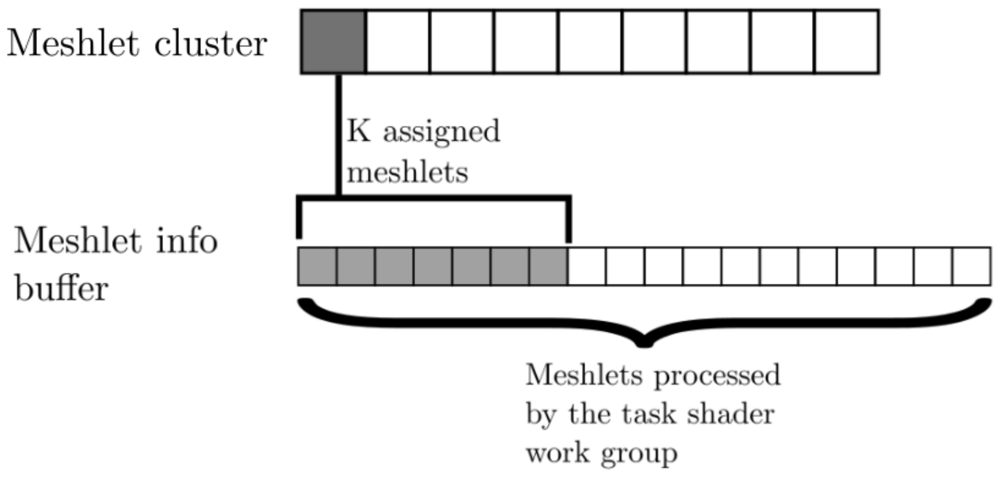


Figure 3.2: Layout of the meshlet cluster buffer and the meshlet info buffer.

Mesh shader

The mesh shader program performs the main task of primitive generation. Each work group consists of 64 invocations, where each invocation generates two primitives (which make up 1 quad) based on the information contained in its respective meshlet cluster. This cluster is selected based on the `gl_WorkGroupID.x` index. This structure is visualized in figure 3.4.

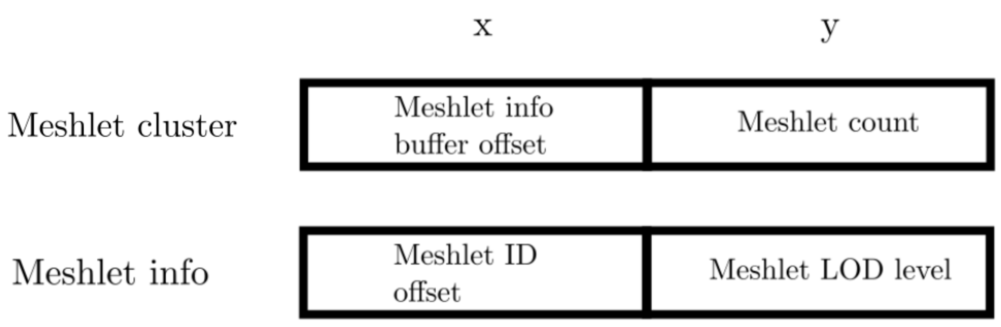


Figure 3.3: Layout of meshlet cluster and meshlet info structures.

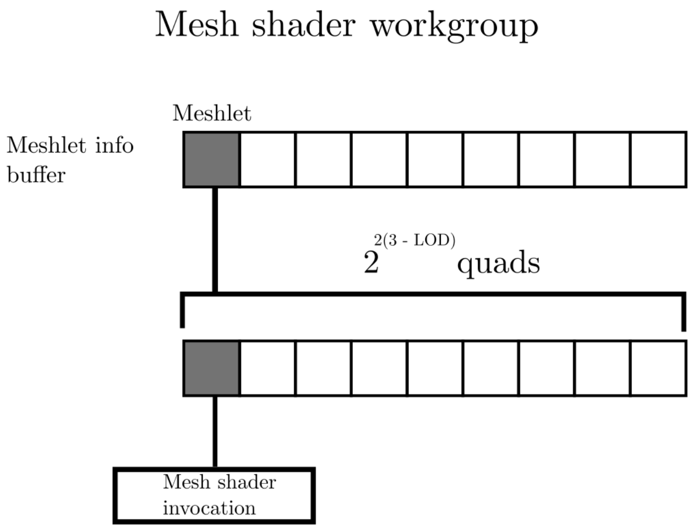


Figure 3.4: Mesh shader work group and invocation structure. The work group processes several meshlets, and each invocation processes a quad from one of these meshlets.

Each meshlet contained in a given meshlet cluster is of the same LOD level. Let $LOD \in \{0, 1, 2, 3\}$ be this LOD level (LOD optimization is further explored in 3.4.2) — the number of quads inside the meshlet Q is then equal to $2^{2(3-LOD)}$. Using Q, for each mesh shader invocation, using its `gl_LocalInvocationID.x`, we determine the meshlet info buffer offset to be `gl_LocalInvocationID.x / Q` and the quad ID to be `gl_LocalInvocationID.x % Q`. This way each meshlet cluster is capable of containing a varying number of meshlets which scales with the LOD level.

Having identified the quad to be drawn by the mesh shader invocation, we calculate the global meshlet ID by obtaining the offset from the `meshletInfo` object and adding it to the global offset included in the task shader payload. We then access the `meshletDescription` SSBO and obtain the necessary heightmap elevation values to finally create the vertices and primitives.

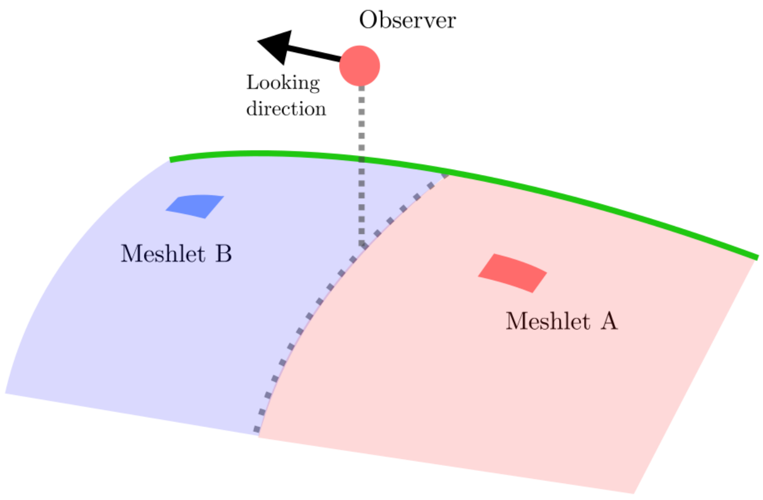


Figure 3.5: Diagram presenting how the observers looking direction dictates that meshlet B ought to be rendered however meshlet A be culled.

3.4 Optimizations

3.4.1 Culling methods

Meshlet and heightmap culling is an important optimization that avoids unnecessary task and mesh shader invocations. Although the methods below describe the methods only for meshlets, the same optimizations are applied to whole heightmaps too, except these are implemented on the CPU whereas meshlet culling is performed on the GPU.

Two cases can be distinguished where culling is viable: when a meshlet is beyond the visible horizon and when a meshlet is behind the observer.

Basic view culling

Frustum culling is the method of culling mesh data that would otherwise be rendered outside of the view frustum[11]. The method implemented in this scenario is different, as it only takes into account whether the meshlet we are trying to cull is behind the observer (figure 3.5), meaning certain meshlets will still be rendered outside of the observers view frustum. This method however is simpler and requires less data required for culling calculations to be passed into the shader programs.

In order to see if the meshlet should be culled, we take the vector $\vec{d}$ being the direction in which the observer is currently looking and the vector $\vec{v}$ = meshletPosition - observerPosition. Next we take the dot product of the vectors and if the result is less than or equal to 0, the meshlet will be culled.

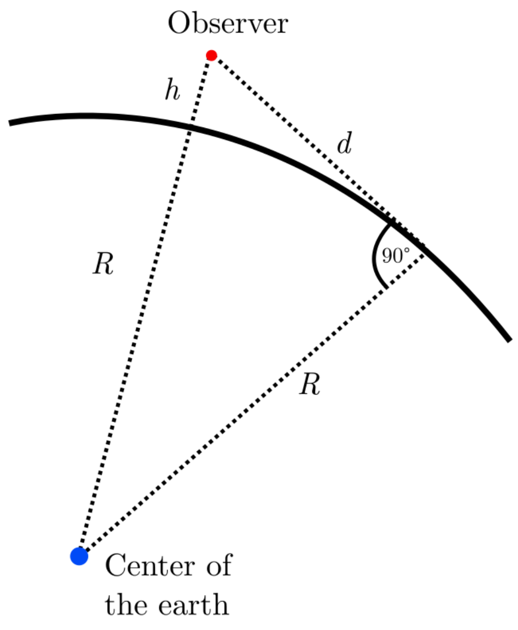


Figure 3.6: Horizon distance diagram.

Horizon culling

To calculate the horizon distance for the observer, only the Pythagorean theorem is needed (figure 3.6).

$$\begin{aligned} d^2 + R^2 &= (R+h)^2 \\ d &= \sqrt{R^2 + 2Rh + h^2 - R^2} \\ d &= \sqrt{2Rh + h^2} \end{aligned}$$

To cull a meshlet, we take the global positions of its four corners and compare the distances between them and the observer with the calculated horizon distance. If all of the distances are longer than the horizon distance, we can conclude the meshlet is beyond the horizon and hence cull it.

3.4.2 LOD rendering

LOD rendering, meaning level of detail rendering, is an optimization including the software scaling a given asset's detail level[12]. For a mesh this often means switching between models of varying polygon count and for textures utilizing different resolutions.

In the case the terrain renderer, the optimization is very natural given the

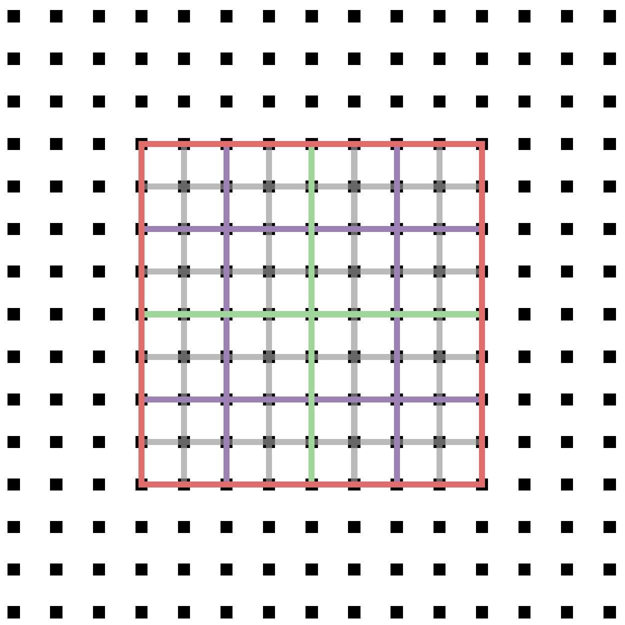


Figure 3.7: Presentation of the squares generated with each LOD level. The colours gray, purple, green and red respectively represent LOD levels 1, 2, 4 and 8.

structure of the meshlets — each meshlet can be rendered in four LOD levels: 0, 1, 2, 3, where with each LOD level quads are constructed from heightmap elevation values separated by 2^{LOD} units (figure 3.7); given that the meshlet's length is fixed to 9 units, such quad building is possible.

The meshlet LOD level is calculated following the formula:

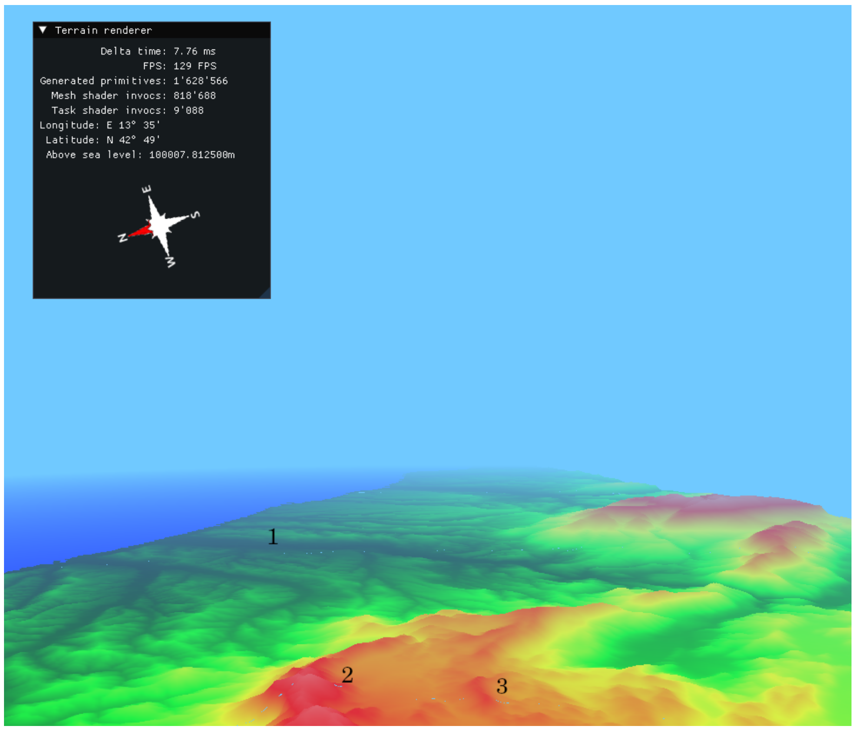
$$dist = \textbf{length}(\text{meshletPosition - observerPosition})$$
$$LOD = \textbf{roundDown}(\frac{\textbf{clamp}(dist, 0, 800)}{250})$$

This causes meshlets further from the observer to be rendered with less details and polygons to improve performance.

Meshlet stitching

When using LOD optimization, one problem that may appear is lack of meshlet stitching (figures 3.8 and 3.9). This is caused when two meshlets neighbouring each other are rendered with different LOD levels. This problem is purely visual and does not affect the performance, however in a commercial use-case addressing these artifacts would be necessary.

One solution is creating transitional meshlets that align one side with the neighbouring lower-LOD meshlet's vertices. This however requires detecting where such meshlets are and which sides ought to be prepared differently which may impact shader performance. In the terrain renderer, this issue is ignored due to the small scale of the problem.



```
▼ Terrain renderer
          Delta time: 7.76 ms
                 FPS: 129 FPS
Generated primitives: 1'628'566
  Mesh shader invocs: 818'688
  Task shader invocs: 9'088
Longitude: E 13° 35'
 Latitude: N 42° 49'
 Above sea level: 100007.812500m
```

Figure 3.8: Artifacts caused by the lack of meshlet stitching can be seen in the labeled parts of the screenshot.

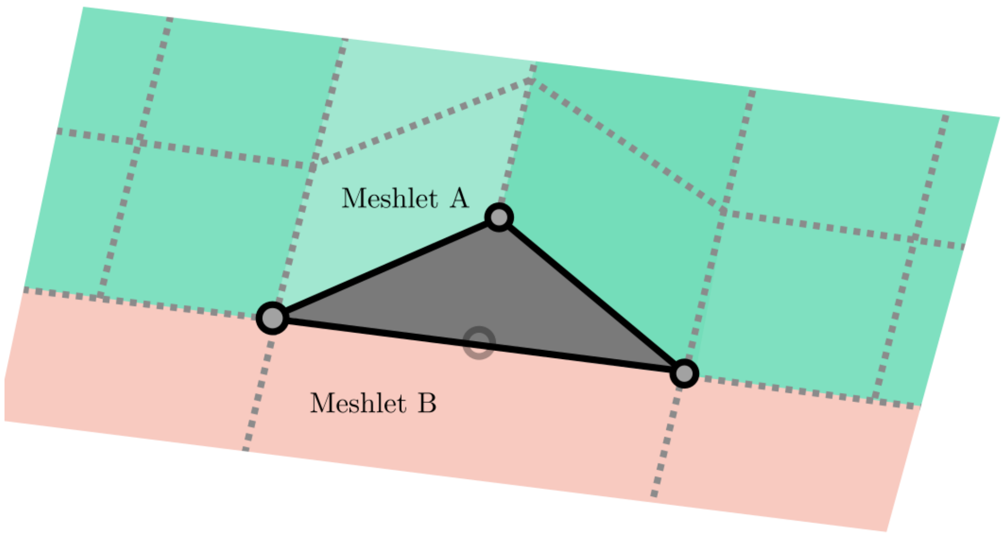


Figure 3.9: A diagram presenting the potential artifact caused by lack of meshlet stitching. Meshlets A and B are neighbouring meshlets of different LOD levels.

Comparison to the traditional pipeline solutions

Using mesh shaders and in turn meshlet-based shader design, LOD rendering is made much simpler and more intuitive than in the traditional rendering pipeline. Elements such as LOD selection in traditional solutions often require implementing compute shaders that would perform the selection. Additionally, alternate primitive generation still requires a separate implementation of the tesselation or geometry shader, whereas, using mesh shaders, the selection and generation is naturally implemented along both the task and mesh shaders.

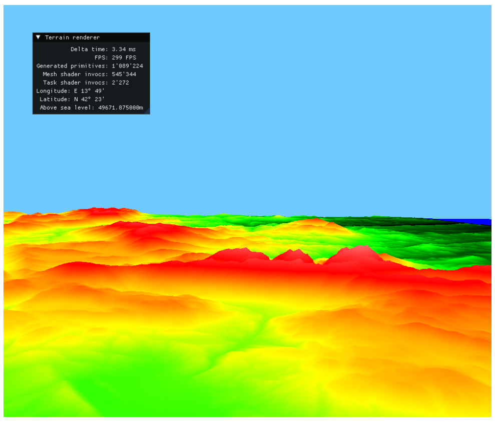
Chapter 4

Performance

The terrain renderer was tested in several cases and scenarios with the performance recorded. The calculated averages are presented in the result tables. All heightmaps are of the SRTM-3 format and were downloaded from `Coverage map viewfinderpanoramas`[13].

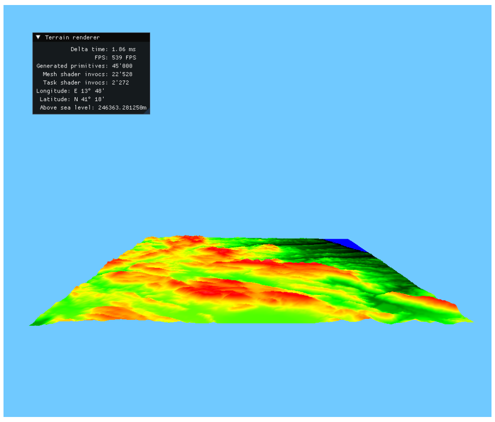
All tests were performed on a Lenovo Legion 5 laptop with a AMD Ryzen™ 5 4600H CPU, an RTX 2060 GPU and 32 GB of RAM.

4.1 Case 1: Single heightmap



```
▼ Terrain renderer
           Delta time: 3.34 ms
                 FPS: 299 FPS
Generated primitives: 1'089'224
  Mesh shader invocs: 545'344
  Task shader invocs: 2'272
Longitude: E 13° 49'
 Latitude: N 42° 23'
 Above sea level: 49671.875000m
```

(a) The heightmap being rendered with the observer in the middle and close to the ground.



```
▼ Terrain renderer
           Delta time: 1.86 ms
                 FPS: 539 FPS
Generated primitives: 45'000
  Mesh shader invocs: 22'528
  Task shader invocs: 2'272
Longitude: E 13° 48'
 Latitude: N 41° 18'
 Above sea level: 246363.281250m
```

(b) The heightmap being rendered from far away with each meshlet at LOD level 3 ($\frac{1}{8}$-resolution)

Figure 4.1: Two captured scenarios including a single heightmap being rendered.

For the single heightmap case, two scenarios were tested:

(a) The heightmap being rendered with the observer in the middle and close to the ground

(b) The heightmap being rendered from far away with each meshlet at LOD level 3 ($\frac{1}{8}$-resolution)

The heightmap contains 1201×1201 elevation values, meaning 2880000 triangles account for the whole heightmap. This also means 22500 meshlets are prepared for this heightmap.

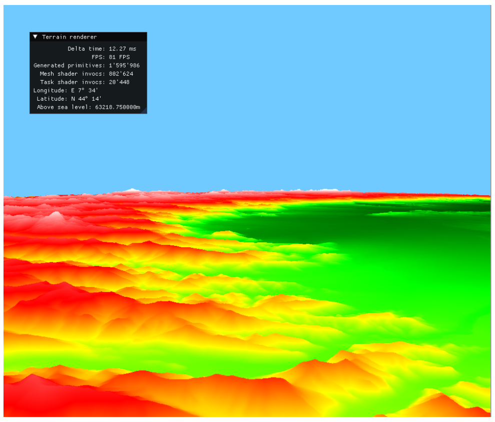
Single heightmap test results				
Scenario	Primitives generated	Task shader invocations	Mesh shader invocations	Render time (ms/FPS)
(a)	$\sim$ 1.00M trigs	2272	~540,000	3.2ms 312 FPS
(b)	45,000 trigs	2272	22,528	1.7ms 588 FPS

4.1.1 Results analysis

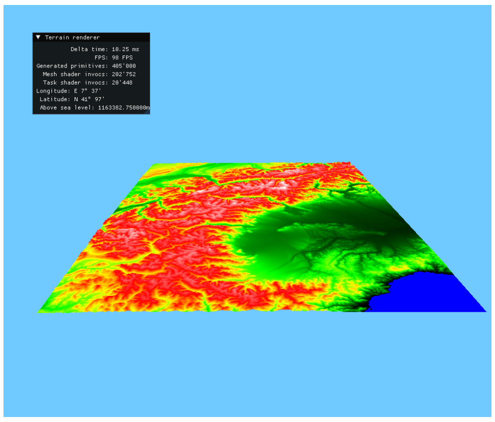
Two observations can be made regarding these results. Firstly, the number of generated primitives is in fact around twice the amount of mesh shader invocations — this is expected as each mesh shader invocation should generate one quad, however it is important to note the program is not generating a significant amount of useless mesh shader invocations.

Secondly, despite the number of mesh invocations being almost 25 times higher in scenario (a) than (b), the performance only reduced by 50%. It will become apparent with subsequent cases however that the performance will depend on several factors, primarily the number of task shader invocations.

4.2 Case 2: Nine neighbouring heightmaps


```
▼ Terrain renderer
          Delta time: 12.27 ms
                FPS: 81 FPS
Generated primitives: 1'595'986
  Mesh shader invocs: 802'624
  Task shader invocs: 20'448
Longitude: E 7° 34'
 Latitude: N 44° 14'
 Above sea level: 63218.750000m
```

(a) The heightmaps being rendered with the observer in the middle and close to the ground.


```
▼ Terrain renderer
          Delta time: 10.25 ms
                FPS: 98 FPS
Generated primitives: 405'000
  Mesh shader invocs: 202'752
  Task shader invocs: 20'448
Longitude: E 7° 37'
 Latitude: N 41° 97'
 Above sea level: 1163382.750000m
```

(b) The heightmaps being rendered from far away with each meshlet at LOD level 3 ($\frac{1}{8}$-resolution)

Figure 4.2: Two captured scenarios including nine neighbouring heightmap being rendered.

For the 9 neighbouring heightmaps case, two scenarios were tested:

(a) The heightmaps being rendered with the observer in the middle and close to the ground

(b) The heightmaps being rendered from far away with each meshlet at LOD level 3 ($\frac{1}{8}$-resolution)

The heightmap names range from `N44E006` to `N46E008`, meaning they represent majority of the Alps mountain ranges. There are 202500 meshlets that need to be processed when rendering the heightmaps.

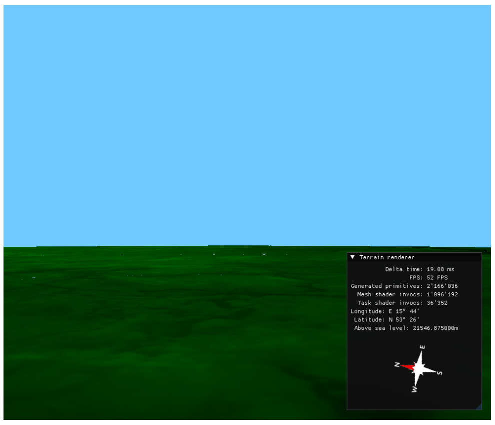
Nine heightmaps test results				
Scenario	Primitives generated	Task shader invocations	Mesh shader invocations	Render time (ms/FPS)
(a)	~1.59M trigs	20,448	~802,000	12.00ms 81 FPS
(b)	405,000 trigs	20,448	202,752	10.60ms 95 FPS

4.2.1 Results analysis

A key observation can be made here based on the results of case 1 and case 2: despite the number of mesh shader invocations in case 2 scenario (b) being less than a half of the invocations in case 1 scenario (a), the performance is 3× worse. This is caused by the significant increase of task shader invocations which are much slower than the mesh shader invocations as their tasks are more complex. This may suggest reducing the number of task shader invocations or reducing the amount of calculation that must be done and moving it to the CPU could be significantly beneficial to the performance.

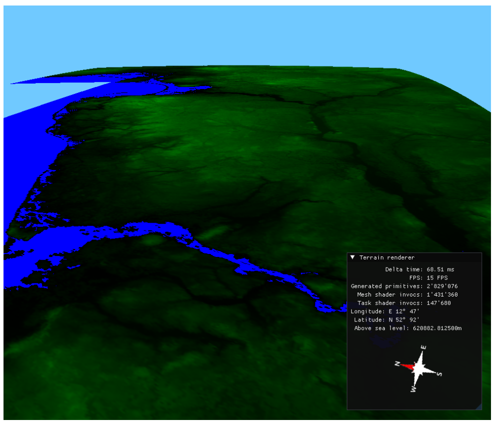
It is also worth noting that despite the number of primitives rising almost 4× (like the mesh shader invocation count) between the cases, the rendering time is stable and only changes by 15%. This again suggests the key role of the task shader invocations and how their count makes a big impact on the performance.

4.3 Case 3: Northern Poland heightmaps



```
▼ Terrain renderer
            Delta time: 19.08 ms
                   FPS: 52 FPS
 Generated primitives: 2'166'036
   Mesh shader invocs: 1'096'192
   Task shader invocs: 36'352
Longitude: E 15° 44'
 Latitude: N 53° 26'
 Above sea level: 21546.875000m
```

(a) Northern Poland being rendered with the observer positioned centrally and close to the ground.



```
▼ Terrain renderer
            Delta time: 68.51 ms
                   FPS: 15 FPS
 Generated primitives: 2'829'076
   Mesh shader invocs: 1'431'360
   Task shader invocs: 147'680
Longitude: E 12° 47'
 Latitude: N 52° 92'
 Above sea level: 620882.812500m
```

(b) Northern Poland being rendered from high above the ground.

Figure 4.3: Two captured scenarios including heightmaps from the northern Poland region.

For the northern Poland heightmaps case, two scenarios were tested:

(a) The heightmaps being rendered with the observer positioned centrally and close to the ground

(b) The heightmaps being rendered from high above the ground

This case makes use of 68 heightmaps, resulting in 1530000 meshlets that may need to be processed at a time.

Northern Poland test results				
Scenario	Primitives generated	Task shader invocations	Mesh shader invocations	Render time (ms/FPS)
(a)	∼2.17 trigs	36,352	∼1.10M	19.08ms 52 FPS
(b)	∼2.83M trigs	147,680	∼1.43M	68.05ms 15 FPS

4.3.1 Results analysis

The observations from case 2 suggested the number of task shader invocations could be making a significant impact on the performance on the program. The 3.5× reduction in render time between scenarios (a) and (b) and the significant increase in task shader invocations appears to support this observation further. Additionally, the majority of these task shader invocations handle LOD level 3 meshlets which account for a lower primitive count and subsequently result in a low task shader invocation to primitives generated ratio.

Chapter 5

Conclusion

5.1 The mesh shader workflow

The process of creating an application utilizing mesh shaders forces the developer to make certain core design choices in terms of mesh handling and rendering. These are natural and intuitive in the case of a terrain rendering program where the heightmaps can be split into regular chunks and rendered separately.

The design based around meshlets allows for a more straightforward approach regarding LOD selection. In commercial software often pre-processing such as LOD selection or frustum culling is performed using compute shaders or on the CPU side. Although the traditional pipeline rendering programs are also often parallelized, this is dependent on a given vendor and their implementation. Mesh shaders avoid this and deliver themselves the compute-like work group architecture.

The compute-like work group architecture of mesh shaders however migrates much of the responsibility regarding GPU warp utilization onto the developer. It is necessary to make choices that make efficient use of GPU warps. Additionally, it is key to also make efficient use of the task shader payload with a standard limit of 16KB.

The migrated responsibility also includes vendor-specific preferences. Unlike the traditional pipeline where extensions are generally optional and may be used for a given vendor's hardware, here the preferred limits regarding primitive output, work group sizes and other key elements of the mesh shaders need to be queried and taken into consideration.

5.2 Terrain renderer problems and solutions

The terrain renderer implemented in this thesis has several problems that if addressed, may increase the performance significantly.

Firstly, as stated in the performance tests and especially case 3 (4.3) results analysis, the number of task shader invocations appears to be the main bottleneck in terms of performance. The task shaders appear to be suboptimal and their significant increase also causes likely significant increases in rendering times. One way task shaders could be utilized more optimally would be by scaling the amount of meshlets per task shader invocation with their LOD levels — for example, a group of 64 LOD level 3 meshlets can be placed in one meshlet cluster. Assuming a task shader work group processes only 320 meshlets and all of them are LOD level 3, only 5 meshlet clusters will be created. This is an important case as LOD level 3 meshlets are the most popular, yet producing only 5 meshlet clusters per 32 task shader invocations is a major waste of potential task shader payload memory.

Another problem that may be noticed with the terrain renderer, is the 1:2 ratio of mesh shader invocations to primitives generated. Each mesh shader work group is capable of outputting 256 primitives and 256 vertices when using the RTX 2060, however reserving a mesh shader invocation for generating only one quad will only produce 128 primitives and 256 vertices in the best case scenario. Generating new vertices per each quad is not optimal however allows for a simpler implementation including the meshlet clusters. An implementation utilizing meshlet clusters that avoids duplicate vertex generation could be possible however, it would require the meshlet clusters to be prepared in compact chunks rather than simple strips. This would ultimately create a hierarchal design of heightmaps, meshlet clusters and meshlets.

Lastly, the shader programs feature branching in certain areas of the shader code, which results in poor utilization of the SIMT (Single instruction, multiple threads) model. In order for shader work groups to perform calculations optimally, single instruction multiple threads cores[14] found on the GPU require the invocations to have a similar or the same instruction set — every branch however, causes invocation instruction splits and increases SIMT thread demand. Avoiding these branches could be done using bitwise operations, alternative calculations, loop unrolling and other techniques.

In summary, the program serves a good starting point and introduction to mesh shader programming, however still requires much work in terms of optimization and better warps usage.

Bibliography

[1] E. Haines, N. Hoffman, and T. Möller, "Real-Time Rendering, Fourth Edition", Chapter 2, pp. 12-18, 2018

[2] C. Kubisch, "Introduction to Turing Mesh Shaders", 2018

[3] NVIDIA, "Vulkan & OpenGL CAD Mesh Shader Sample", Example mesh shader project, 2018

[4] C. Kubisch, "Using Mesh Shaders for Professional Graphics", 2020

[5] V. Zoutman, "Turing Mesh Shaders", Chapter 1, 2020

[6] C. Kubisch, "Mesh Shading for Vulkan", 2022

[7] Khronos Group, Mesh shader draw calls in Vulkan Specification, 2024

[8] Cezary Czubała, https://github.com/Cez02/LOD-terrain-renderer-thesis, Terrain renderer source code, 2024-2025

[9] NASA and NGA, "The Shuttle Radar Topography Mission (SRTM) Collection User Guide", 2015

[10] Google, Shaderc project source code, 2015

[11] E. Haines, N. Hoffman, and T. Möller, "Real-Time Rendering, Fourth Edition", Chapter 19, pp. 835-836, 2018

[12] S. Marschner, P. Shirley, Fundamentals of Computer Graphics, 5th Edition, Chapter 22, pp. 630, 2021

[13] C. Hormann, J. de Ferranti, Coverage map, 2011

[14] J. Luitjens, S. Rennich, "CUDA Warps and Occupancy", 2011