Procedural Terrain Rendering using Mesh Shading

Etienne Renoult
Université Grenoble-Alpes
INRIA Grenoble

Supervised by: Thibault Tricard

I understand what plagiarism entails and I declare that this report is my own, original work.
Name, date and signature:

Abstract

In this report we propose an implementation of a procedural terrain generation and rendering using the recent Mesh Shading pipeline. We used work groups of threads proposed by the said pipeline to take advantage of the GPU's parallel power. The heightmap texture is generated through a Compute Shader, using patches to draw regions of the image in parallel. The geometry is also generated using patches, each patch corresponding to a work group and representing a part of the geometry. We made an analogy between the work group ID and the position of the patch in the geometry.

We also implemented a distance-based level of details, in order to improve the performances. We made sure of visual coherence between the patches by some extra process in the geometry generation. The goal of our work is to state if the said pipeline can provide visually accurate renderings, evaluate if it is efficient in terms of performances and how the level of details influences those performances.

1 Introduction

Large virtual worlds are multiplying. Whether in video-games, movies or even in mobile applications, those worlds tend to be as wide and realistic as possible. The multiplication of open-world games, and wide world rendering in general is part due to the evolution of hardware, but especially to computer scientist who came up with very efficient optimization techniques for terrain rendering such as **occlusion culling**, **tessellation**, **Level Of Details** (LOD) and others. Repesentation of terrains can find a lot of applications as entertainment, but also simulations, urbanism and more.

The procedural aspect of rendering enable more realism, but also requires less work for artists or programmers in general which can focuse on global shapes of geometry. It also enable features such as animation for example in water rendering.

Improvement of realism, performances and scale of such representation can reduce the costs of production, while allowing to better immersion, or better simulation results, and even more accessibility to those virtual worlds for systems that can not benefit for latest-generation hardware. However, achieving these advancements requires to manipulate complex geometry, while ensuring scalability and flexibility due to the diversity of environments to represent and the diversity of devices and hardware, for which rendering and optimizations techniques should be possible to adapt.

1.1 The Mesh Shading Pipeline

Most of those techniques are dating from more than a decade, and even though some new techniques has been uncovered and designed, the old ones are still mostly used because of their good trades between simplicity and efficiency. Moreover, technical advancements on the subjects made by private companies are not shared publicly.

However, in recent years, the topic has regained the attention of engineers and researchers thanks a new technology introduced by **Nvidia** in 2018 : the **Mesh Shaders**, [Kubisch, 2018].

This technology is quite similar to compute shaders, because it provides a complete and simplified control over all the GPU's available cores, thus reduces the locality of geometry operations. This way, the programmer can take advantage of the full parallel power of the hardware to generate and modify the geometry. While the tessellation, culling and geometry steps were separated along the rendering pipeline, needing communications of data between each steps, the mesh shading pipeline is simpler than a more traditional pipeline, thus allowing these steps to be brought together in a single one for more control. Moreover, the flexibility of the Mesh shader step keeps classical optimization techniques such as LOD and culling suitable.

The goal of this work is to investigate the efficiency of the mesh shading pipeline for procedural terrain rendering, by implementing a generation and rendering pipeline with some of the classical optimization techniques and then evaluate its performances on terrains with different scales and resolution.

1.2 Context And State Of The Art

Most of the experiments regarding Mesh Shading performances are related to real-time terrain rendering, for example by transposing the terrain tessellation from the classical graphic pipeline to the Mesh shading pipeline [Santerre *et al.*, 2020]. It leads to better performances while allowing to increase the maximum tessellation capacity by 2. Using the maximum amount of tessellation provided by this technique does not lead to better visual results, and does not worth the performance loss, but it is not a real problem knowing that such levels of tessellation are never used.

Another way of increasing the performances by using Mesh Shaders has been explored by [Hedberg and Bartel, 2022], who worked on representing terrain as nodes of a quadtree and simplify those nodes. Unfortunately, with the concurrent binary-tree-traversal algorithm used, the mesh shading pipeline does not yield significant improvement compared to the traditionnal graphic pipeline. With that said, the chunked Level-of-Detail rendering is more stable with mesh shading than with Instance Render (regular graphic pipeline), and performance could have been further improved with the use of advanced culling.

Finally, terrain rendering performances can be greatly improved with Mesh Shader because it allows to get rid of some bottlenecks from the traditional graphic pipeline. For example, a dynamic allocation of vertices can be avoided using a vertex pool, for which the cache locality can be maintained, [Englert, 2020]. Also, the geometry data can be sent to the GPU incrementaly, and triangle-based primitives can be computed by the GPU as meshlets, which is the core of the mesh shading concept.

Our approach regarding the LOD disctretization of terrain regions, is based on quad-tree and level of details computation from [Lindstrom *et al.*, 1996], excepts that each region of a terrain will be the same size in the geometry. The difference between such regions is the number of triangles, thus, details, that they display.

Even if it already exists several implementations of terrain rendering using mesh shaders, they focus more on efficiency of optimization and modificiation of geometry instead of pure generation. Thus, it is interesting to build a pipeline for procedural terrain generation, with optimizations and minimal loss of visual quality, in order to evaluate the performances of the Mesh Shading pipeline.

2 Contribution

One of the simplest way to represent terrains in 3D is to create a grid of vertices along x and y axis of the scene, and to displace them along the z axis. This grid is called a Mesh, and our Mesh will have a size of $N * N$ vertices, as in the example in Figure 1.

Generally, the displacement of the vertices is performed according to an **heightmap texture**, vertices in the grid matches pixels in the texture, and the brightness of the pixel determines the height of the vertex. This is what we used for the terrain generation.

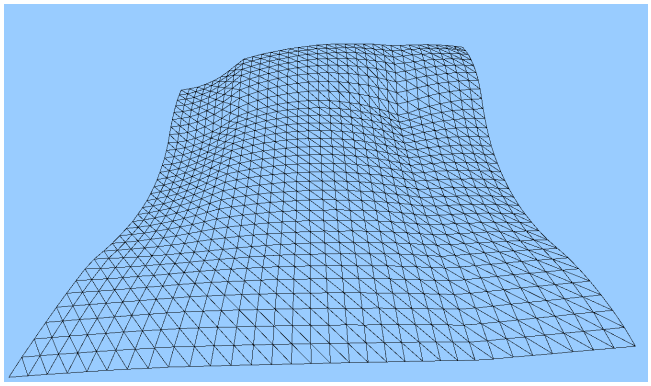

Figure 1: Example of a mesh of 33 by 33 vertices

2.1 Our Pipeline

Now that we know how we want to generate the geometry, we have to procedurally generate a texture with natural and organic aspects. We deciced to generate it using Perlin noise [Perlin, 1985].

The Perlin Noise For Heightmap Texture Generation

This noise is very basic, and quite old, but it is very easy to understand, to execute, and most of all, it is fitted for parallel computing, which is mandatory in our case. Moreover, it provide controls over its appearance thanks to the frequency parameter.

We actually used an enhanced version of the Perlin noise, using an hash function for the gradient vectors instead of a permutation table, [McEwan *et al.*, 2012]. We also combined multiple instances of the noise with itself using the Fractal Brownian Motion as suggested by [Perlin, 1985], in order to create a more organic, less soft texture. Moreover, the texture generation would work with any noise that can be computed in parallel.

Indeed, the texture was generated in parallel, using a Compute shader, which is the real first step of our pipeline.

The Compute Shader

Before generating the texture we have to determine its resolution. In our case, the number of pixels on which we will perform the Perlin noise will correspond to the number of vertices in the mesh.

The Compute shader allows us to distribute the work among work groups, in which it will be distributed again among threads. The work groups can be identified by global X, Y and Z coordinates, and threads inside a group are identified by local x, y, and z coordinates. Threads can also be identified by global coordinates, but we did not use them. A representation of such parallelization is shown in Figure 2

In order to compute and generate the Perlin noise texture in parallel, we had to divide the texture in **patches**. As said before, the texture is a square mesh with a size of $N*N$ vertices. Then every patch is a square of size $(N/p) * (N/p)$ vertices, p being the number of patches along the size of the square. Thus, each patch corresponds to one work group, with coordinates $\{X, Y, Z\}$, with $Z = 1$. We will not consider the Z

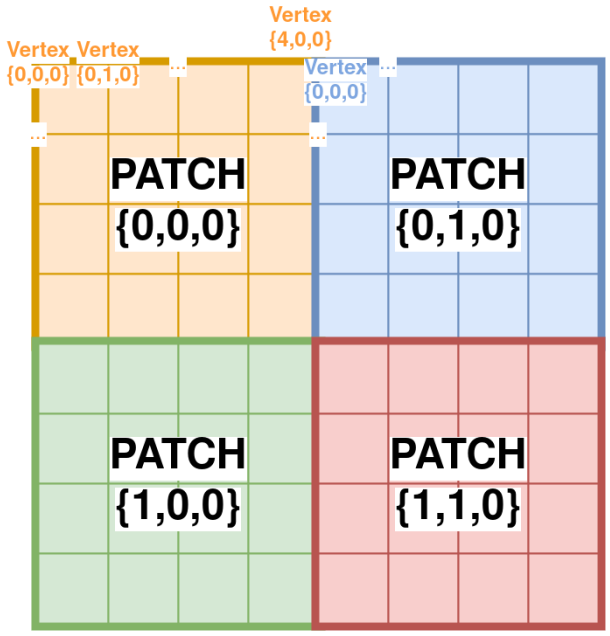

Figure 2: Example of texture with 2 by 2 patches (work groups), each patch having 4 by 4 vertices (working threads), forming a grid of 4 by 4 squares. Patches can be identified through their global ID, while threads have a local ID.

coordinate of patches anymore since we generate the geometry from a 2D texture, and a 2D mesh. For a patch with coordinates $\{X_p, Y_p\}$, each thread of global coordinates $\{x, y\}$ corresponds to the pixel of coordinates $\{X_p * x, Y_p * y\}$ normalized in the final texture.

Note that the frequency, in the Perlin noise parameters is independant from the number of patches we want to generate.

The Mesh shader

Once the heightmap texture is generated, we can start generating the geometry. A Mesh shader behave the same way as a Compute shader, excepts that it can output geometry primitives. The parallelization of work groups and threads is the same, using 3D coordinates too.

Thus, we used the same number of patches p to generate parts of the final terrain mesh, with this time each thread corresponding to one vertex. Finally, we used a normalization of the vertices coordinates in x and y, to get the corresponding color in the texture, and move the vertex along z depending on the brightness of the color. The color brightness being normalized, we multiply it by a arbitrary factor so that z would vary between 0 and the maximum height chosen. A result of such generation can be observed in Figure 3.

Unfortunately, we are limited by the hardware-implemented constraints of Mesh shaders for the number of primitives we can output. The maximum amount of triangles that can be generated by a single work group is 128. Knowing that, we can deduce that the maximum resolution

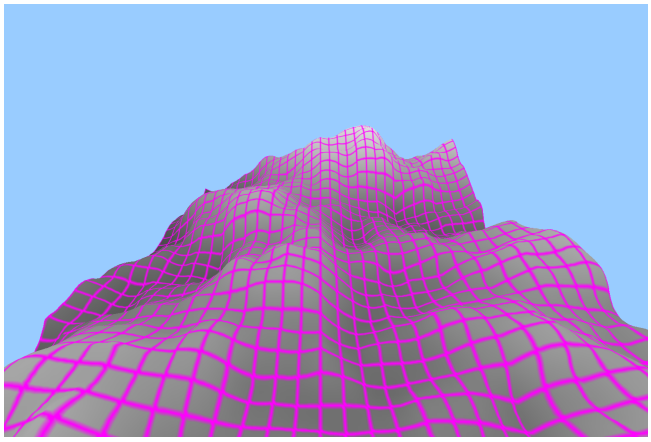

Figure 3: Results of terrain generation using Perlin noise, Magenta lines represents borders of patches, containing several triangles

of the patch is 9 by 9 vertices, which will form a mesh of 8 by 8 squares, each one composed of 2 triangles, which leads to 128 triangles.

The Distance-Based Level Of Details

In order to save computing resources, we would like to generate less triangles. One way to do it is to replace a detailed geometry requiring a lot of small triangles by a simpler one with bigger triangles, when the said geometry is far away from the camera. The goal of such optimization is to reduce the number of primitives while keeping a decent representation of the geometry. Sometimes the difference can not be spotted because the detailed geometry is already too far away from the point of view to display all of its details.

We implemented a simple distance-based level of details (LOD) at a patch level, which mean each work group will compute its own LOD. The computation happens in the Mesh shader step of the pipeline. The level of details will changes the number of vertices in a patch, without changing its total size. Since the maximum number of vertices per patch is $9*9$ as said earlier, the LOD would make number of vertices on a side of a patch vary from 2 to 9. The less detailed patch possible has only 2 triangles.

Our LOD is computed depending on the distance between the camera and the patch, being represented by one set x, y and z coordinates. Those coordinates are the ones from the 3D point in the middle of the patch in x and y, and at half of the maximum height possible in z. Each patch is placed in the scene depending of its X and Y work group IDs, which means that each patch can compute its own position in the scene, allowing the LOD to be computed in parallel, at patch level. An example of geometry generated by the Mesh shader using distance-based LOD for patch resolution is available in Figure 4

2.2 The Pacth Borders

One of the main problems of having different LOD in the same 3D shape as continuous terrain, is to keep geometry co-

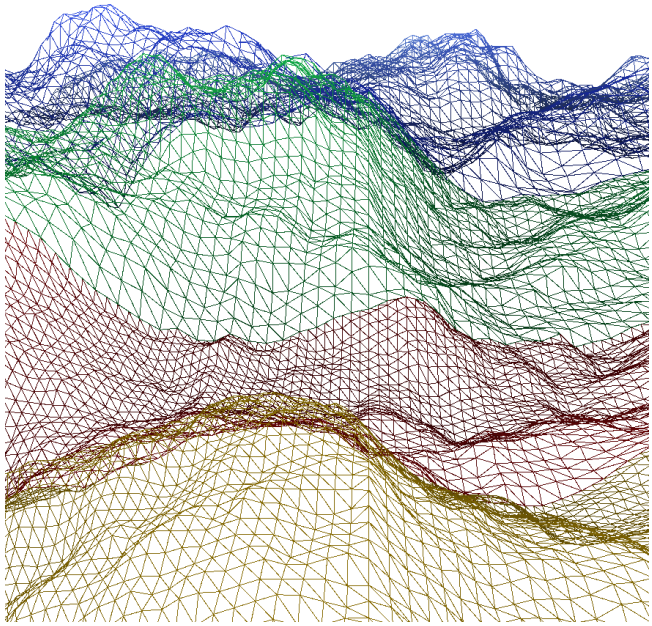

Figure 4: Terrain generation with LOD. Yellow triangles represent patches of $9 * 9$ vertices, red patches are $5 * 5$, green ones are $3 * 3$ and blue patches are $2 * 2$ vertices

herence between parts of the geometry with different amount of details. Indeed, borders of neighbor patches with different LOD does not always match together, because of the different discretization of the curve they represent. It results in holes in the geometry appearing at those borders.

There are several way to fix such holes, but in most cases, we try first to have more control on the different levels of discretization between adjacent patches. Actually, the most convenient way to discretize patches is to do it following powers of 2 in for the numbers of squares along a patch's side. Thus, instead of having patches with side of n vertices, we keep only patches with sides $2^n + 1$ vertices, with $n \in \mathbf{N}$, and $n \in [0, 3]$. Using the number of vertices on a patch's side, the possible resolutions of patches are described in Figure 5.

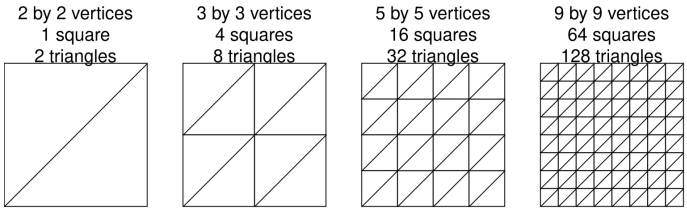

Figure 5: Possible discretizations for patches

Once we make sure that adjacent patches are different from at most 1 level of details, then it is quite straightforward to fix the holes between the said patches. With 2 patches p_1 and p_2, with $LOD(p_1) > LOD(p_2)$, we would have to modify the border that p_1, which is more discretized, has in common with p_2, in order to make it match with the less discretized border. We know that between 2 vertices of this border in p_2, we can find one from p_1. We will just move those p_1's vertices along the z-axis to make them match the line between the 2 vertices of p_2 (the average of their z coordinate). The hole fixing is explained in Figure 6

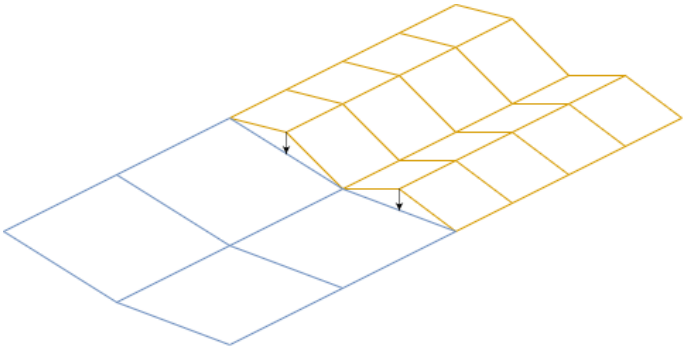

Figure 6: Schematic of the border fixing. The vertices of the most discretized border that are between 2 vertices of the less discretized one along the border change their z coordinates in order to match with the segment of the other border.

If we represent vertices along p_1's border by indexes from 0 to $LOD(p_1)$, we can find the vertices to move by taking the odd ones.

Since each patch can compute its own LOD only knowing its work group ID coordinates, each patch can easely compute the LOD of its neighbors by shifting those coordinates by 1. In this way, a patch can know if it needs to fix one or several of its borders, without needing other information than its coordinates X and Y.

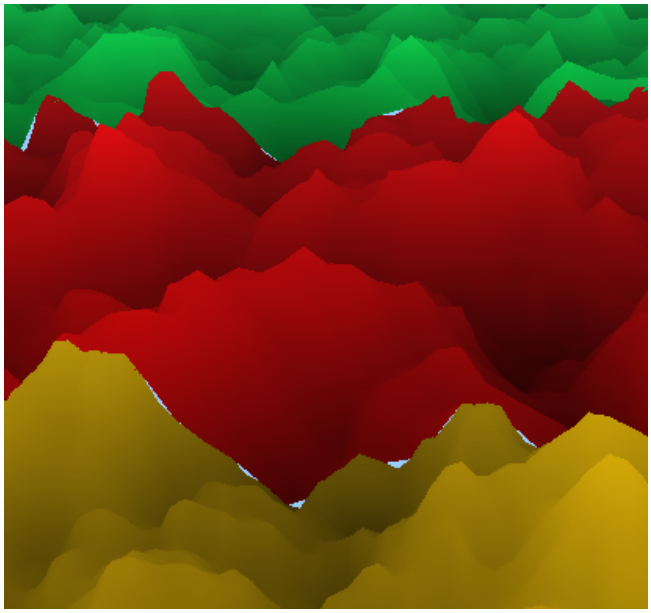

Figure 7: Mesh with holes at patch borders

Results of such operations on a terrain generation changes the aspect of the render and make it cleaner, as you can spot the difference between Figure 7 and Figure 8.

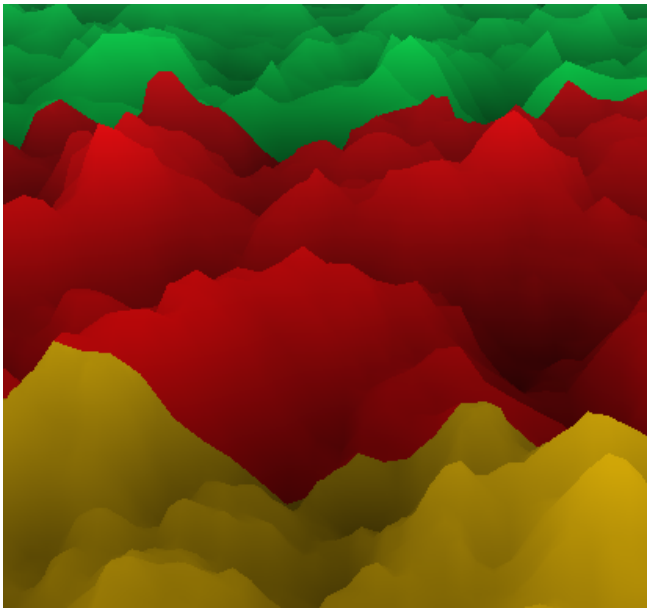

Figure 8: Mesh without holes at patch borders

2.3 The experiments

In order to evaluate the performances of this terrain generation with Mesh shading, we decided to execute the pipeline on terrain of different scales, with different LOD settings. Note that the number of octaves parameter for the Perlin noise generation stays the same between each stage. However the texture size (number of pixels) that it will have to generate, and the number of cells in may change, according to the size of the terrain we want to generate. The number of cells changes proportionally to the size of the texture in order to keep the same frequency. Those parameters changes in order to keep the same aspect for the terrain.

Settings

As described at the introduction, the generated terrain is a squared mesh, with its sides composed of P patches. We evaluated the performances in FPS for $P \in [64, 640]$ with increments of 64. At each increment, we compared the performances for terrains with every patch at maximum LOD, minimum LOD in order to have an upper bound and a lower bound of performances, and for terrains with distance-base LOD with and without fixing borders, in order to calculate the influence of such operations on the global performances. You can see below the number of patches and primitives for each evaluation.

We are performing all of those evaluations on a Intel Xeon 4210R CPU, with a 64 GiB RAM and an Nvidia RTX A4000 GPU.

# of patches on the terrain's side	64	128	256	512	640
PATCH COUNT	4,096	16,384	65,536	262,144	409,600
2 points per patch :					
TRIANGLE COUNT	8,192	32,768	131,072	524,288	819,200
FPS	6,264	1,707	355	56	30
9 points per patch :					
TRIANGLE COUNT	524,288	2,097,152	8,388,608	33,554,432	52,428,800
FPS	4,979	1,548	322	53	28
LOD					
FPS	5,798	1,665	351	56	30
LOD + fixed borders					
FPS	5,206	1,545	336	54	29

Table 1: Results of the evaluations

Results

If we look at the number FPS regarding the number of patches on the terrain's side (Table 1), we can observe that the performances are quite similar between different level of discretization, with the same number of patches. The largest differences between 2 points and 9 points level of details is when we evaluate them with 64 patches. The more we ask for patches, the more we tend to low FPS, almost below real-time at the end.

We can also notice that for the same number of triangles, performances are completely different if we ask for a different amount of patches. We tried with another discretization to make sure that the only difference between the 2 renders is not the number of triangle asked, but the number of patches.

Terrain side (in # of patches)	64	512
Total number of patches	4,096	262,144
Triangles per patch	9	2
Total triangle count	524,288	524,288
FPS	4,979	56

Table 2: Table of evaluations for patches of different resolutions, but same amount of triangle asked

Even though the performances of such a terrain rendering pipeline are quite good (real-time in most cases), and allow to display a decent amount of triangles within a very short delay, the implemented optimizations did not seem to have a real influence of the global performances. Even if we considerably reduced the number of triangles asked compared to a render with level 9 LOD, we did not increased the FPS.

3 Conclusion

We proposed a rendering pipeline allowing to take advantage of parallel power of the GPU for both procedural heightmap texture generation and geometry generation based on this heightmap. This procedurally generated heightmap could be easely be replaced by another noise generation, since the geometry generation is in an independant step of the pipeline.

Moreover, since the heightmap is generated at each frame, feature such as animations, or further generation of new patches are enabled.

However, we cannot state of the efficiency of the distance-based LOD optimization, since the performances seems to be more related to the number of work groups invoked than the number of triangles requested. This might be due to the fact that the number of threads allocated by the Mesh shader is fixed. Which means that no matter how many primitives we want to output, the Mesh shader will only use the threads that it needs but all other will be idleing.

One way to make sure of it would be to implement another LOD technique, and terrain discretization, based on quadtrees, where every patch would be discretize the same way, but will cover different size of the terrain. Instead of same-sized patches with different triangluations, we would have same-triangulated patches with different size. In this way, the total size of the terrain would not depend anymore on the number of patch asked, but on the level of details.

Since in this case we might see the difference of performances with different amount of primitives asked, this work could then be combined with other optimization techniques such as occlusion culling, in order reduce even more the number of triangles, and highlight this difference.

References

[Englert, 2020] Matthias Englert. Using mesh shaders for continuous level-of-detail terrain rendering. In *ACM SIGGRAPH 2020 Talks*, SIGGRAPH '20, New York, NY, USA, 2020. Association for Computing Machinery.

[Hedberg and Bartel, 2022] Martin Hedberg and Alexandre Bartel. Rendering of large scale continuous terrain using mesh shading pipeline. Master's thesis, Umeå University, June 2022.

[Kubisch, 2018] Christoph Kubisch. Introduction to turing mesh shaders. at `https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/` (2024/04/22), 2018.

[Lindstrom *et al.*, 1996] Peter Lindstrom, David Koller, William Ribarsky, Larry F Hodges, Nick Faust, and Gregory A Turner. Real-time, continuous level of detail rendering of height fields. pages 109–118, 1996.

[McEwan *et al.*, 2012] Ian McEwan, David Sheets, Stefan Gustavson, and Mark Richardson. Efficient computational noise in glsl. *arXiv.org*, April 2012.

[Perlin, 1985] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, jul 1985.

[Santerre *et al.*, 2020] Benjamin Santerre, Masaki Abe, and Taichi Watanabe. Improving gpu real-time wide terrain tessellation using the new mesh shader pipeline. In *2020 Nicograph International (NicoInt)*, pages 86–89, 2020.