Using Mesh Shaders for Isosurface Extraction

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Master of Science in Computer Science

at

The University of Waikato

by

Hamish Elliott

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2022

Abstract

This thesis investigates the use of the Mesh Shader graphics pipeline, utilising in particular its ability of parallel and dynamic choice of geometry, to gain computation efficiencies in realising the Marching Cubes isosurface extraction algorithm. The thesis details how the Marching Cubes algorithm was mapped to the Mesh Shader pipeline, to form a baseline implementation. In addition, two heuristic-based variants are presented that reduced the GPUs workload, but at the potential cost of compromising accuracy. In evaluation of the work, a comparison of the baseline Mesh Shader pipeline implementation is made with a previous compute-shader based implementation. We also present a set of experiments that investigate the speed and accuracy of the two heuristic-based variants.

Acknowledgements

This endeavour would not have been possible without Mr Bill Rogers and Professor David Bainbridge for providing advice, encouragement and mentoring throughout this research. I would also like to thank fellow students for our shared engagement and comradery.

Contents

Chapter 1

Introduction

The release of the NVIDIA Turing micro-architecture in 2018, part of the NVIDIA series of graphics processor unit architectures, saw several new features introduced, along with the extension of some existing features. These features span, but are not limited to, hardware accelerated ray tracing units, tensor cores and mesh shaders. Hardware accelerated ray tracing units are used for the ray tracing graphics pipeline, which uses hardware designed for ray traversal through defined sets of geometry. Tensor cores are used for fast hardware accelerated, low precision matrix operations, primarily useful for machine learning purposes. While tensor cores were introduced in the (previous) Volta micro-architecture in 2017, it was not until Turing that they became widely available in consumer grade graphics processors.

The last of the major innovations in Turing, and the focus of the work reported in this thesis, is the mesh shader pipeline, a partial alternative to the traditional graphics pipeline. The mesh shader pipeline provides a completely different way of supplying geometry for subsequent processing in the pipeline. In previous improvements in the traditional graphics pipeline, they have been in the form of amending new stages. The new stages in the mesh shader pipeline take the form of two new shader stages: the task shader and mesh shader stages. These stages allow the programmer full access to the parallelism of the GPU in processing or generating geometry.

NVIDIA have provided extensions to expose mesh shader functionality in the OpenGL, Vulkan and Direct3D 12 graphics APIs. AMD subsequently made their own implementation of the mesh shader pipeline, first appearing in their release of the RDNA2 micro-architecture in 2020. However, their implementation is only exposed in the Direct3D 12 graphics API. AMD have not released extensions for OpenGL or Vulkan. Instead, they have opted to wait for the standards groups to provide hardware agnostic extensions.

The traditional graphics pipeline has a fixed system for inputting data to be drawn. This fixed system means that shader invocation requires input data

to be available in GPU memory prior to a draw call and the only method of workload scaling within the pipeline is by using a geometry shader or the tessellation stages. The geometry shader processes one primitive at a time to inject a new set of primitives. The tessellation stages a parameterised built-in algorithm to subdivide the input geometry.

Given the full access to the parallelism of the GPU in processing or generating geometry that the mesh shader pipeline provides, in comparison to the traditional graphics pipeline, there is a wider variety of ways to increase the efficiency of rendering using not only workload scaling, but workload culling as well. One use case for mesh shaders highlighted by NVIDIA is for workload culling on a triangle list mesh. In this approach, segments of the mesh—termed meshlets in NVIDIA parlance—are dynamically culled within the draw call. Because mesh shaders have their general purpose compute capabilities, rendered meshes can be split into coherent segments of geometry and culled at a time, eliminating subsequent work from being passed down from an earlier in the graphics pipeline.

Using the Stanford bunny as the exemplar, Figure 1.1 shows an illustration of its mesh being divided into coloured segments. After loading the bunny, it is divided into segments of coherent sets of triangles. Each segment is given the range of directions each triangle is facing. Segments are then culled when the full range is determined to be facing away from the viewport camera. This process happens inside of the draw call during pipeline execution. In comparison, backface culling is the closest thing the traditional pipeline offers, where individual triangles are considered in isolation, and those triangles which face away from the viewport camera are culled. The latter technique, therefore, is not capable of exploiting spatial coherence as can be done in the former.

1.1 Problem Description

As new hardware features, such as NVIDIA's mesh shader pipeline are released, they allow for new opportunities to improve on old methods or create new ones. The new mesh shader pipeline introduces a General Purpose Graphics Processor Unit (GPGPU) compute-like approach to providing geometry to the rasterizer, giving the ability to synthesise geometry in the GPU within draw instructions. Marching Cubes, as an example of an isosurface extraction algorithm, appears as an appealing algorithm to implement using the mesh shader pipeline due to its ability to be parallelised easily, suitable for the parallel design of the GPU.

This thesis focuses on using the mesh shader pipeline for geometry synthesis as this topic is an under-researched area in the literature. In particular, we

Figure 1.1: The Stanford bunny represented in meshlets. Reproduced from Kubisch (2018).

have chosen to study the highly used Marching Cubes algorithm. Marching Cubes is an important practical algorithm and that it is often used to discover sparse structures, allowing us to explore opportunities for work culling as well as construction.

We hope to exploit characteristics of the mesh shader pipeline to implement the Marching Cubes algorithm. To achieve this, we consider the limitations of previous implementations and informed by this, develop an implementation of the Marching Cubes algorithm that takes advantage of the features the mesh shader pipeline affords. In doing so, this will provide insights—both in terms of strengths and weaknesses—that can be applied more generally to geometry synthesised algorithms on the GPU.

Because Marching Cubes is an algorithm which examines a density function defined over a volume and creates a triangle list mesh around around volumes of high density. Such an algorithm makes for a strong candidate to use, to investigate processing volume data and synthesising geometry to be drawn.

1.2 Synopsis

The structure of this thesis is as follows. In Chapter 2, we cover the background to two key topics: the Marching Cubes algorithm, and mesh shaders. It also reviews previous related work in both of those areas and considerations from

previous implementations. Design and Implementation (Chapter 3) describes how the mesh shader Marching Cubes implementation developed for this work was devised and the tools used to create it. The following chapter reports on the performance tests undertaken using the new mesh shader method, before concluding with a summary of our findings in Chapter 5.

1.3 Scope

Given the time-constraints to the thesis, the implementation work and performance experiments were undertaken using a single GPU architecture: the aforementioned Turing architecture, by NVIDIA. There is no reason to believe there would be any impediment in developing a version that runs on the AMD architecture, although—as the Vulkan API was used in this work—the developed code would need to be ported to Direct3D 12.

For rendering the result of the Marching Cubes algorithm, as will be seen in the figures provided in Chapter 5, flat shading was used. Inclusion of textures and smoothed normal vectors into the implementation would have improved the rendering quality. However, it was not part of the Marching Cubes algorithm itself, so it was similarly omitted due to time-constraints.

As a writing convention, this thesis makes use of the word “we” to refer to the work of the sole author.

Chapter 2

Background

Some prerequisite understanding of both the Marching Cubes algorithm and the mesh shader pipeline is required to be able to apply the topics this chapter covers to the rest of this thesis. In Section 2.1 we introduce the broader subject of isosurface extraction before focusing our attention on the Marching Cubes algorithm. Input for Marching Cubes can be empirically gathered volume data—common in medical applications—or else synthesised. We use the latter for our performance experiments in Chapter 4, and so Section 2.2 provides an overview of noise functions commonly used to provide synthesised data to the Marching Cubes algorithm. This equips us with fine-grained control over characteristics of the input data we can run throughout mesh shader pipeline implementation and testing of Marching Cubes.

Section 2.3 (Graphics Processor Architecture) gives an overview of the hardware which the new graphics feature, mesh shaders, is executed on. This provides insight as to how code is executed on the GPU and how to design programs optimally. This leads to a section on mesh shaders, going over details about what they are and what some potential use cases they cover. Finally the results of a literature review is presented. It shows some limitations that a previous GPU-based implementation of Marching Cubes has, what Marching Cubes provides and the kinds of advantages the mesh shader pipeline has to offer.

2.1 Isosurface Extraction

Isosurface extraction algorithms are used to create geometry to visualise boundaries in volumetric data. The origins of defining geometry by filling volumes can be traced back to Blinn (1982), giving the idea of creating objects out of metaballs. Blinn's method creates surfaces from mathematical definitions of objects, and shows the surface between inside and outside of the object.

Marching Cubes is an algorithm which rose to prominence from its ability

to create triangle models of constant density surfaces from 3D medical data (Lorensen & Cline, 1987). In addition to organ visualisations (Y. Wang, Zhong, & Hua, 2020), this algorithm is well-known and widely applied (Newman & Yi, 2006), for instance, in fluid simulation (W. Wang, Jiang, Qiu, & Li, 2012).

A more recent application of isosurface extraction algorithms is to visualise volumes generated from the Instant Neural Radiance Field algorithm (NeRF) (Müller, Evans, Schied, & Keller, 2022). NeRF uses a set of images from a scene to generate volume data, to which the Marching Cubes algorithm is then applied to generate a 3D model of that scene.

In summary, input data for isosurface extraction algorithms might be scan data of real objects, volumetric fluid simulation, a computed density function or any other method of obtaining density volumes. The Marching Cubes algorithm divides the volume of interest into a regular grid of cells, and requires that the input data provides or allows computation of a density at each point on the grid. Each cell's set of corners is examined to determine whether and where the surface passes through the grid cell. The triangle list configuration for that cell is decided, depending on the combination of corners that are inside the structure being outlined.

Our work makes use of noise functions to provide computed density information, as described in Section 2.2.

2.1.1 Marching Squares

To help explain our use case for Marching Cubes in 3D, we present here a 2D example. The 2D version of this algorithm is called "Marching Squares". The goal of the Marching Squares algorithm here is to find approximate outlines for the cut out shapes. Figure 2.1 gives an overview of the steps which Marching Squares takes to construct a surface boundary. Figure 2.1a first shows an area with shapes filling in subsections of the areas. This is akin to a sheet of paper with shapes cut out of it (a square and a circle in Figure 2.1a). The areas in white still have paper and the areas in black denote the sections where the shapes have been cut out. Moving to Figure 2.1b, the area has been divided into cells on a grid, with the corners of each cell being examined in Figure 2.1c, corners that lie on the paper being shown in white and corners in holes being black. The transition to Figure 2.1d shows how the base cases shown in Figure 2.2 are used to show the part of the approximate outline (if any) in each cell. So, using the base cases, each cell has its corners examined for whether it resides on paper or it has been cut out built from the chosen base cases, then chooses the base case which represents the cell. From there, Figure 2.1e shows the rough outline of the area which has been cut out and leaves the final image in Figure 2.1f. For the sake of illustration of the algorithm, the grid size

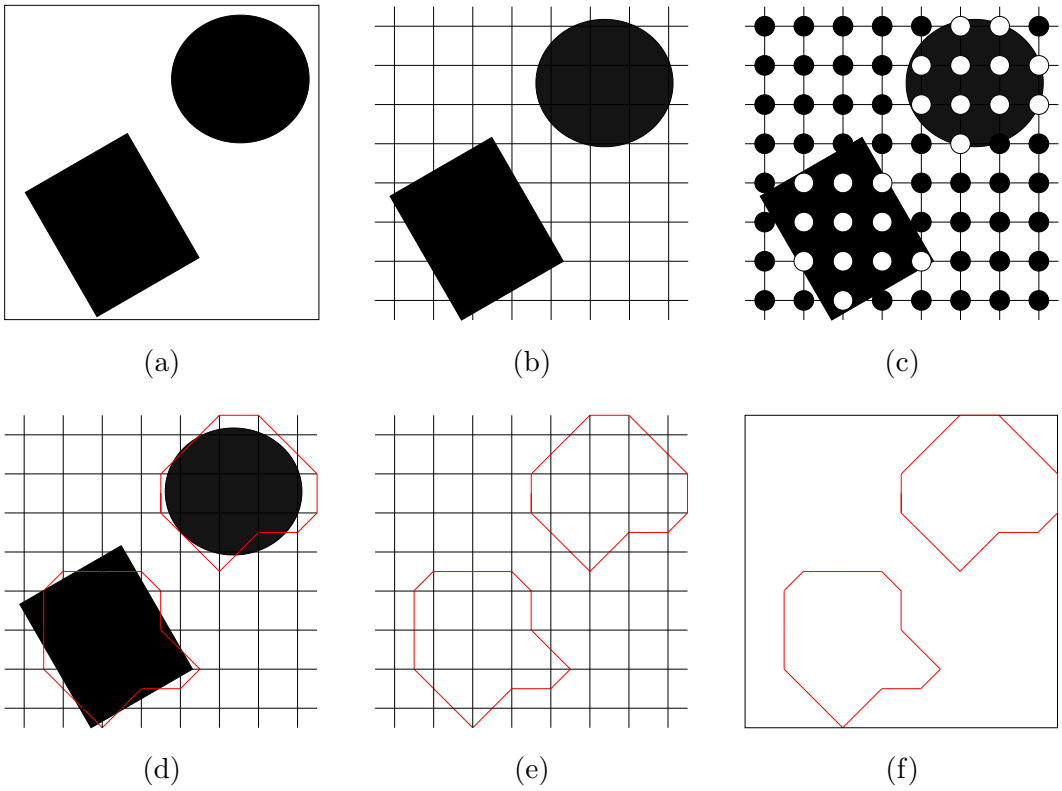


Figure 2.1: Steps through Marching Squares.

selected was intentionally coarse. The effect of this is shown in Figure 2.1d, where the bottom right edge of the circle does not overlap the corner of the grid cell, causing the corner to be recognised as not being cut out and showing the contour in the surface boundary. Another effect of this is the top and the left of the rectangle having contours in the boundary. In practice, using the algorithm at a higher resolution gives more accurate outlines.

The 2D example in Figure 2.1a assumes that we start with a knowledge of which points are inside and outside of the areas of interest. In fact, most real examples start with density information, in which we have a density value we have associated with each point on the surface, and synthesises a boundary between areas of high and low density. Figure 2.3 illustrates the situation. In this case, we have a density value associated with each point on the plane, depicted as a vertical displacement.

Note that even though Figure 2.3 is shown as a three-dimensional scene, this is still a two-dimensional problem to which Marching Squares applies. The Marching Squares algorithm decides whether a point is inside or outside an area, depending on whether the density function is above or below a chosen threshold. The red plane in Figure 2.3 is set at the height of the threshold. Figure 2.4 shows an example of a mesh generated in a 2D context. The red ribbon depicts the boundary where height displacement changes between being

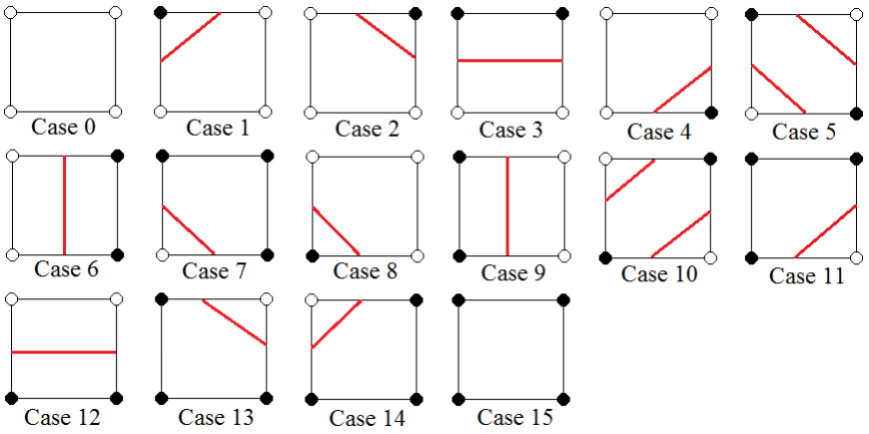


Figure 2.2: 16 topological base cases of Marching Squares. Black vertices have value greater than or equal to the threshold. White vertices have value less than the threshold. Reproduced from Gong and Newman (2013).

below and above the threshold. The smoother ribbon is caused by a higher resolution.

2.1.2 Marching Cubes

The three-dimensional version of the algorithm works similarly to Marching Squares, with the square grid cells of Marching Squares replaced by cubes and the density function giving a value at each point in space, rather than at each point on a plane. An example of such a density function might be the temperature at each regular point within a volume. The result of the 3D version is an (iso)surface in three-dimensional space.

In the 3D case, since there are eight vertices in each cube and once a threshold has been applied, two states being either inside and outside, there are $2^8 = 256$ ways a surface can intersect the cube. In terms of implementation of the algorithm, Figure 2.5 shows the set of 14 unique configurations of triangles. The total of 256 configurations come from inversions, reflections and rotations of each of the 14 unique configurations. By enumerating these 256 cases, we create a table to look up surface-edge intersections, given the labelling of a cube's vertices. The table contains the edges intersected for each case (Lorensen & Cline, 1987).

The algorithm works by examining each cell in a regular grid. Each cell's triangle list configuration is determined by examining the combination of the corners in each cell to determine whether they are inside the dense structure or not, followed by selection of the matching entry in the look-up table. The cell's set of triangles are then included in the mesh. With an understanding of the 2D case, it can then be extended to a 3D volume and mesh being

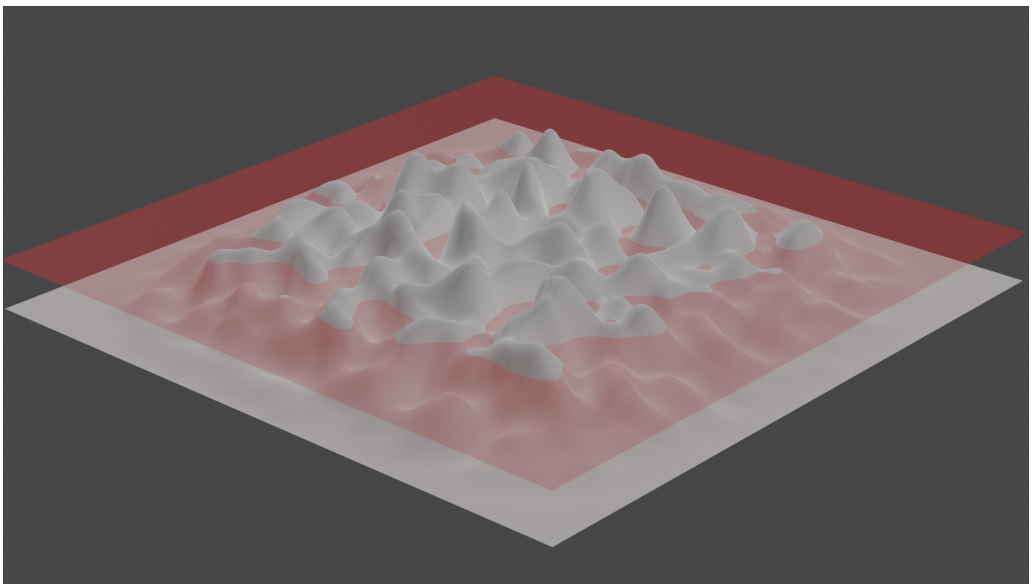


Figure 2.3: 2D plane height-displaced by Perlin noise with a red plane representing the threshold.

generated. An extension to the 3D mesh is the importance of a triangle's winding order. For presentation to the triangle rasterization stage of graphics pipelines it is conventional to order the vertices of a triangle so that they appear in clockwise order when observed from outside the object they define. This is used for backface culling of triangles. Therefore, it is important for the base cases to be constructed in a way that is conscious of each triangle's winding order.

The Marching Cubes algorithm requires the input to be a regular grid of scalar values across a total volume. Another way to think of this is a three-dimensional area of voxels (Xiang, Choi, Lin, & Savarese, 2015), with each voxel having a set of eight corners, being shared with a neighbouring voxel. When the Marching Cubes algorithm runs, the eight corners to each voxel is examined. The resulting configurations of each cell contributes to the overall mesh being generated.

2.2 Noise Functions

The Marching Cubes algorithm uses an input of a regular grid of scalar values. An example of obtaining data is some form of computed data. Noise functions provide a pseudo-random scalar value when given a vector position. Using this, we can obtain results from noise functions across the entire volume for Marching Cubes.

Perlin noise (Perlin, 1985) provides smooth and coherent values which span over a space. In two dimensions, Perlin noise generates values for each point

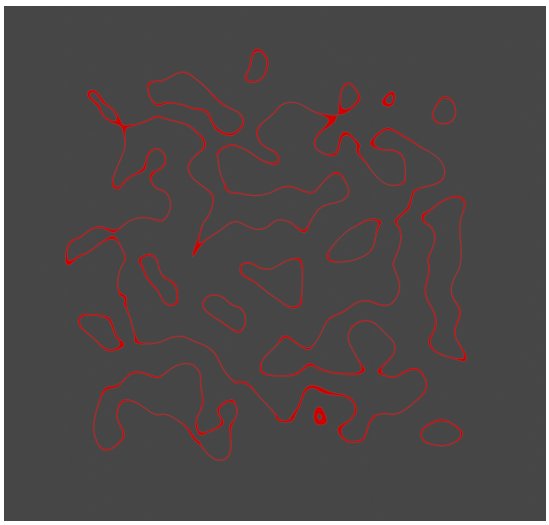


Figure 2.4: Ribbon showing the boundary between values above and below the threshold.

on a plane; in three dimensions, it generates values at each point in a volume. It does so by randomly generating vectors at points on a regular grid, then interpolating them throughout the grid cells. This interpolation is weighted by the polynomial $3t^2 - 2t^3$, giving a smooth appearance and avoiding a mach banding effect. Finally, the algorithm calculates the dot product between the interpolated vector and the closest vector along the regular grid to yield the final result of the noise function. In doing so, it yields a pseudo-random result that changes smoothly through the given space.

The noise algorithm is later improved upon (Perlin, 2002a) by choosing from a set of 12 pre-determined vectors and interpolating by the formula $6t^5 - 15t^4 + 10t^3$. The reason for the change in the interpolant formula is to avoid artefacts. This is because the new formula has zero in the first and second derivatives at $t = 0$ and $t = 1$, which avoids discontinuities. The second improvement, in speed, is the result of choosing from the set of simple predetermined vectors with coefficients of 0, +1 and -1; so avoiding multiplications in the dot product calculations.

Simplex noise was then created (Perlin, 2002b) as a replacement for the original noise algorithms. In this method, a simplex is chosen from a simplex grid; a simplex grid being a simplical tesselation of N-space (Gustavson, 2005). In 3D, the shape which composes a simplex grid is a tetrahedron. The corner of each simplex in the simplex grid has a radial influence around it. The particular simplex is chosen by taking the integer part of a point's co-ordinates at each dimension. From there, influence from each corner and neighbouring simplexes are then summed together to create the noise value for the given point.

The provided benefit of noise functions is they allow for controlled charac-

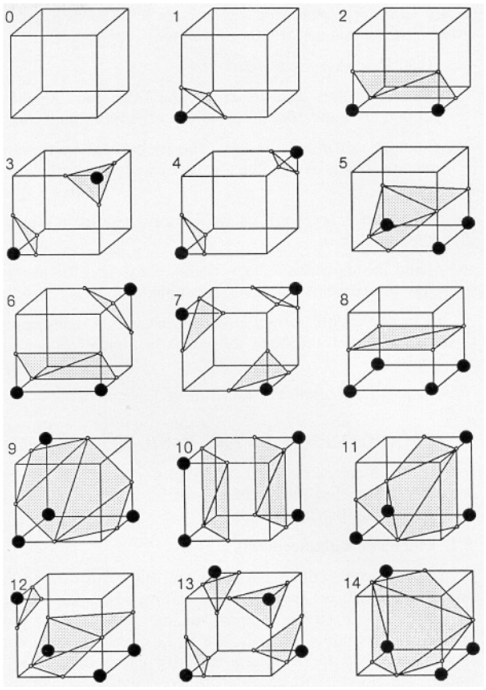


Figure 2.5: Triangulated Cubes. Reproduced from Lorensen and Cline (1987).

teristics of the input data. This is primarily beneficial in two ways. The first is to be able to control the input data for experimentation. The other benefit is that they are used to generate procedural surfaces such as terrain and cloud effects.

2.3 Graphics Processor Architecture

When creating implementations for a Graphics Processor Unit (GPU), understanding its architecture and how it runs programs is crucial. Here, from the NVIDIA family of GPU architectures, each of them has their own set of features. Examples of these include NVIDIA Pascal, Maxwell and Kepler. The GPU architecture we are going to focus on is NVIDIA Turing.

We specifically look at the Turing TU102 GPU. Figure 2.6 gives a schematic overview of this GPU. It shows a collection of Streaming Multiprocessors (SMs), L2 Cache, memory controllers and PCI Express interface. More specifically, it contains the GigaThread Engine which is responsible for dispatching thread groups, 512 KB of L2 cache and a set of Graphics Processing Clusters (GPCs), each containing a dedicated raster engine and six Texture Processing Clusters (TPCs). A TPC includes two SMs, which workgroups are scheduled on. An SM is illustrated in Figure 2.7. SMs contain 64 CUDA Cores, eight Tensor Cores, a 256 KB register file, four texture units and 96 KB of L1 cache/shared memory (NVIDIA Corporation, 2018). Different GPU variants

Figure 2.6: Turing TU102 Full GPU with 72 SM Units. Reproduced from Kilgariff et al. (2018).

such as the TU104 and TU106 retain the same basic architecture, however, they are each scaled down for different grades of performance (NVIDIA Corporation, 2018).

Although graphics cards are used for hardware accelerated graphics workloads, they can also be repurposed for general parallel computation. GPGPU compute is the use of a GPU to do general purpose scientific and engineering computing (Ghorpade, Parande, Kulkarni, & Bawaskar, 2012). Computation using GPGPU is designed to take advantage of the GPU's massively parallel design. NVIDIA CUDA is an example of a platform which offers GPGPU compute. Massively parallel workloads, in this context, could refer to up to thousands of threads. Although in the GPU's hardware this is true, in practice, workloads are split into collections of 32 threads. In CUDA, these groups of 32 threads are referred to as "warps" and are the work units which are scheduled onto an SM. Each thread in a single warp executes the same instruction simultaneously by an SM.

Each SM has 64 warp slots, allowing 64 warps to be active simultaneously, their execution being interleaved allowing work to be done for one warp while others are waiting for results from a unit on the GPU. Examples of this are having one warp wait on a memory fetch while another uses the math units.

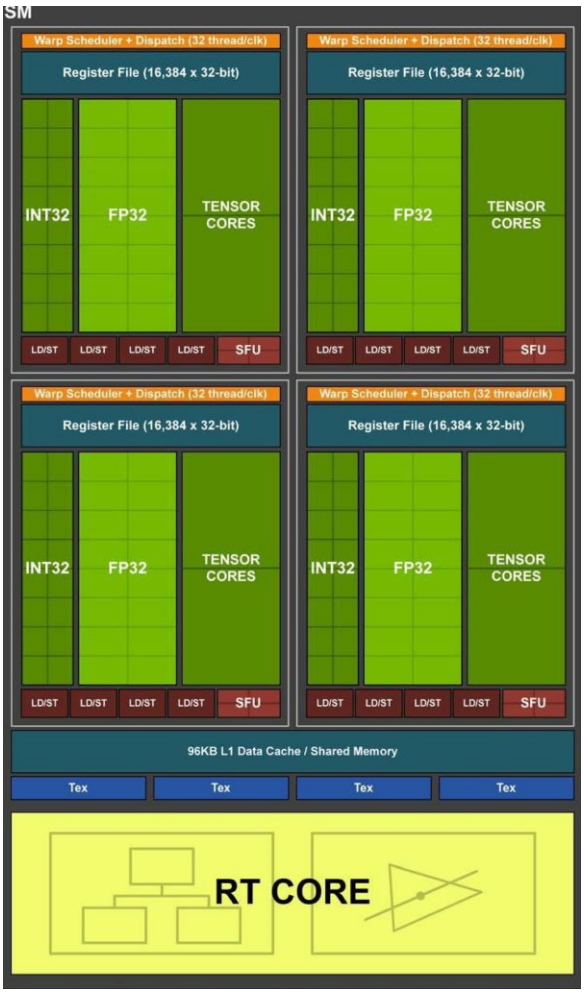


Figure 2.7: Turing TU102/TU104/TU106 Streaming Multiprocessor (SM). Reproduced from Kilgariff et al. (2018).

Interleaving execution of active warps is the principal behind latency hiding (Volkov, 2016), which provides increased parallelism when properly taken advantage of.

Although a given SM has 64 warp slots, the actual number of concurrently executing warps could be inhibited by a given unit on the GPU being used at full capacity. This could be due to the SM's arithmetic units being at capacity, the register file being full, or too many requests to memory.

The Turing architecture contains a new set of features such as Ray Tracing Acceleration for real-time applications, Deep Learning Super-Sampling (DLSS) which uses an AI inference for upscaling images in real-time, Variable Rate Shading for dynamically deciding shading fidelity, Multi-View Rendering for providing render surfaces for canted HMDs (non-coplanar displays) and the new mesh shader pipeline (NVIDIA Corporation, 2018).

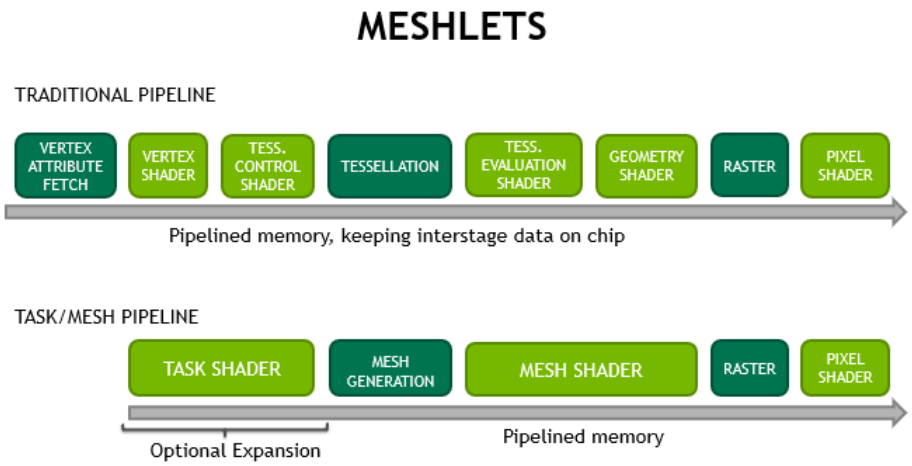


Figure 2.8: The mesh shader pipeline when compared to the traditional graphics pipeline. Reproduced from Kilgariff et al. (2018).

2.4 Mesh Shaders

Figure 2.8 shows an overview of the traditional graphics pipeline and the new mesh shader pipeline for comparison. The traditional fixed function vertex processing graphics pipeline was originally modelled for hardware accelerated graphics cards. This ran by fetching vertices, launching a vertex shader thread for each vertex, possibly modifying geometry with tessellating stages or geometry shaders; then rasterizing and generating display pixels in the the fragment shader stage. Vertex shaders are used to transform vertices by using matrices or other data provided by the user. The programmer only has the perspective of processing one thread's data at a time in this shader. Tessellation shader stages focus on subdividing geometry, with that geometry being passed to the geometry shader. The geometry shader then processes single primitives into small sets of new primitives. Finally, the rasterizer generates pixels from the geometry which are then processed by the fragment shader.

The mesh shader pipeline is a new type of graphics pipeline proposed by NVIDIA, introduced in their Turing GPU architecture (Kubisch, 2018). The mesh shader pipeline introduces two new stages called the task shader and mesh shader stages, which can be described as "compute like" in their capabilities. They are designed to replace the traditional set of geometry processing stages in the graphics pipeline. This means that the new stages take responsibility of replacing vertex input, the vertex shader, tesselation stages and geometry shader stages.

The mesh shader and task shader stages run similarly to compute shaders, where the programmer is given complete control over groups of threads. Because these shader stages derive from how a compute shader runs, some context

in how they run helps with understanding mesh shaders and how the GPU operates as a whole.

Compute shaders are simple, generic programs that run on the GPU. They have access to the GPU's textures, buffers and other resources in its memory, and are designed to be run in large groups of threads. This is suited for execution on GPUs as the nature of their hardware is designed for highly parallel workloads. The large group of threads is split into smaller blocks of a user-defined size. This size is typically set to a multiple of the number of threads in which the GPU's architecture is designed to work with, however this can be adjusted for optimal occupancy on the GPU. These smaller blocks of threads are logically to be considered "workgroups". A number of these workgroups is dispatched by a single instruction from the CPU.

A dispatched set of workgroups is then organised into the executing unit on a single SM. On NVIDIA hardware, these are referred to as warps. This means that the programmer must be aware of the fact that threads in a warp are executed in parallel and any form of branching results in inefficiency in execution.

Mesh shaders and task shaders work similarly to compute shaders. They have the same parallel model of execution and concept of workgroups, with functionality given to each shader for use in a mesh shader graphics pipeline. This new pipeline configuration allows for new optimisation or processing methods that were not possible before. As noted by Kubisch (2018), in terms of where there are limits, these may be different to that of compute shaders, although no specifics are given in his article.

The mesh shader's responsibility is to provide geometry for further processing in the rasterizer and successive stages. Each mesh shader workgroup has a buffer in shared memory associated with it, that contains vertex position, other attribute data attached to each vertex, indices into the vertex buffer to describe the list of primitives and the number of primitives. The set of attributes for each vertex, number of vertices and number of indices is defined by the programmer and is fixed once the shader is compiled.

The geometry held in one of these buffers is referred to as a "meshlet"—just a small section of mesh. Valid sizes are imposed by the shader compiler depend on what hardware it is being run on. Because mesh shaders execute similarly to compute shaders, the method of populating the meshlet is deferred to the programmer—whether it be loaded from resources bound to the shader or completely fabricated during shader invocation. This control comes with the caveat that the responsibility for correctness and optimisation lies with the programmer. Familiarity with topics such as SM occupancy and latency hiding are imperative to creating an optimally performing mesh shader. On

the plus side, this does mean that decisions around the configuration of the mesh can be made in the shader before being written into the meshlet. The total amount of shared memory in an SM is limited to 96 KB, and so a large buffer per workgroup limits the number of workgroups can be scheduled on an SM at any one time. This can be an important factor when trying to take advantage of latency hiding (Volkov, 2016).

Task shaders are an optional component in the mesh shader graphics pipeline. The CPU can directly dispatch mesh shader workgroups, or it can dispatch task shader workgroups where each task shader has the responsibility of launching mesh shader workgroups. This is done by having each task shader workgroup be associated with a number representing the number of mesh shader workgroups to be launched. Task shader workgroups also have a buffer in shared memory for their output, which is accessed by mesh shader workgroups launched by that particular task shader workgroup. The programmer decides which attributes are in the task shader's output buffer. Use cases for task shaders include, but are not limited to, culling specific workloads (i.e. invisible portions of a mesh) and scaling the amount of mesh shader workgroups to allow for more geometry (i.e. tessellation).

Because the mesh shader pipeline comes with limited functionality on its own, it is important to use the pipeline in conjunction with other features to make useful mesh and task shaders. Mesh and task shader functionality revolves around resources belonging to workgroups. Therefore, it is important to use other features for making workgroups cooperatively fill their buffers. Examples include providing access to mesh resources through GPU memory, thread synchronisation using atomic variables and barriers or subgroup intrinsics.

Back in Chapter 1 (Figure 1.1), we briefly described an example by NVIDIA that utilizes the mesh shader pipeline to efficiently render a model of the Stanford Bunny, by culling spatially coherent segments of the model that cannot be seen from the currently rendered viewpoint. Having reviewed how the mesh shader pipeline works, we are now in a position to describe in more detail how this is achieved.

First, on the CPU, the model's geometry is broken into segments, and uploaded to the GPU. For the purposes of illustration in the example, a different colour has been assigned for each of these segments/patches. Each segment corresponds to a meshlet. The mesh's order of indices has been organised for reusage of vertices across meshlets, to provide cache coherence when accessing the geometry (Kubisch, 2018). As a consequence of this, the segments appear as spatially coherent patches.

An advantage of mesh shaders is that they allow for algorithms which were

not possible using the traditional pipeline. For example, a task shader can be used to cull entire meshlets which it has determined are not visible, such as meshlets where every triangle within the meshlet is facing away from the camera. This is precisely what has been done in the NVIDIA Stanford Bunny example, and eliminates work from being done later in the pipeline, reducing workload in the mesh shader, primitive culling, rasterizer, fragment culling methods and fragment shader stages.

Marching Cubes is particularly suited for parallelisation, as each Marching Cubes cell can be processed individually (Dyken, Ziegler, Theobalt, & Seidel, 2008). Because of this, mesh shaders appear to be a suitable candidate for running the Marching Cubes algorithm to generate a mesh, due to the ability to dynamically choose geometry during draw instructions.

2.5 Related Work

Related literature is drawn from the areas of: Marching Cubes and mesh shaders. In reviewing previous literature about those areas, we highlight pertinent topics and how they relate to an implementation of Marching Cubes using mesh shaders as well as aspects of these topics to apply later in this thesis.

A GPU implementation of the Marching Cubes algorithm by Tlatlik (2019) was made using compute shaders for isosurface extraction and the traditional graphics pipeline for rendering, implemented with the Direct3D 12 API and C++. It uses the simplex noise algorithm to create volume data in one compute shader, generating noise for every point and saving it to GPU memory. An implementation of Marching Cubes, is run in a second compute shader stage. Each thread works on a single cell of its total space and then writes the generated isosurface geometry into another buffer, again to GPU memory. That buffer is filled serially starting from its lowest memory address, with addresses being claimed as needed when threads generate triangles. A single atomic counter on the GPU is used to coordinate allocation of addresses by threads.

There are two potential bottlenecks in this solution. One is the amount of time needed to synchronise the atomic counter when populating the vertex and index buffers. Atomic variables are implemented in the hardware and are used for synchronisation in parallel systems. This counter must be incremented and synchronised by every thread that is running on the GPU, causing threads trying to access this counter to wait on each other. The second bottleneck relates to VRAM throughput. Because each of the compute shaders and the graphics pipeline must read from and write to GPU memory, the speed of GPU

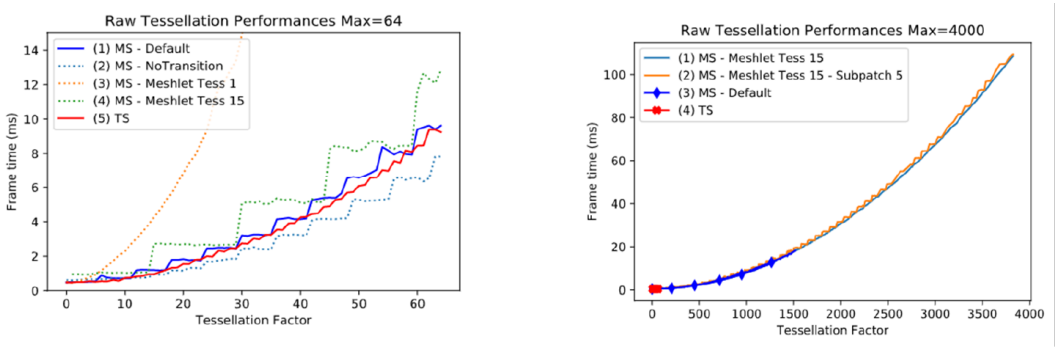
memory is an important factor for the overall speed of the implementation. There is no reflection on the impact of these performance issues in the work by Tlatlik (2019).

Because mesh shader stages provide the GPGPU-like control over parallelism and mesh information is given directly to the rasterizer, we expect to see some form of performance improvement in a mesh shader implementation. This is because it bypasses the requirement for storage to GPU memory and synchronisation through a single atomic counter. It also offers possibilities for workload culling methods, and it runs the entire process of generating a volume and a mesh in a single pass. This is in contrast to the compute shader implementation. Instead of having three separate passes, where a compute shader generates volume data, a separate compute shader creates the surface and a subsequent call to the traditional graphics pipeline to draw the generated mesh.

Mesh shaders are a relatively modern features of recently released GPUs, meaning there has been a limited amount of research published concerning its use. More specifically, no prior work using the mesh shader pipeline to implement Marching Cubes was found. A broader review of research involving the mesh shader pipeline was undertaken to develop an understanding of previous strategies used to implement other algorithms taking advantage of the new pipeline. The result of this review found there has been progress in areas such as skeletal animation (Englert, 2020) and tessellation in terrain rendering (Santerre, Abe, & Watanabe, 2020).

Englert's work on skeletal animation details a technique which localises vertices into meshlets that share common bones. Another technique shown in this work is saving the mesh generated by a mesh shader to GPU memory for subsequent usage through the traditional pipeline. This is a demonstration of leveraging the GPGPU compute capabilities of mesh shaders in conjunction with the traditional pipeline while simultaneously using the generated mesh immediately in the mesh shader pipeline. It eliminates the need for regeneration of the mesh in every draw call.

The work by Santerre et al. using tessellation for terrain rendering is another example of how mesh shaders can be used for applications not previously possible. It is an effective demonstration of how mesh shaders can be used to scale workloads. It also shows how they can exceed the maximum number of tessellations the fixed function hardware unit for tessellation provides. Figure 2.9 shows how the mesh shader implementation exceeds tessellation shader's maximum factor of 64, although there is a slight performance cost. This is possible because the use of task shaders to programatically launch mesh shaders allows very large scaling of workload during execution.



(a) (b)

Figure 2.9: Shows the raw tessellation performances of different algorithms by measuring the time taken to tessellate, respectively for (a) and (b). Measured performance up to a tessellation factor of 64 for comparison with tessellation stages shown in (a). Demonstration of scalability using mesh shaders in (b) (Reproduced from Englert (2020))

Workload culling is also an area where the mesh shader pipeline has the ability to improve overall performance in rendering. In terms of a general strategy, the task shader takes the lead in the decision making process, typically analysing spatio-temporal bounds on segments of a mesh. It uses those spatio-temporal vertex bounds of its assigned vertices, view frustum culling and back-face culling to reduce workload. This provides the means of culling meshletes which will not be visible in the end result. Applications of these techniques gives a performance improvement (Unterguggenberger, Kerbl, Pernsteiner, & Wimmer, 2021).

With understanding of both the mesh shader pipeline and strategies used in previous techniques, for Marching Cubes and mesh shaders independently, we can now devise a solution which implements Marching Cubes and create culling techniques using the new mesh shader pipeline.

Chapter 3

Design and Implementation

To devise a mesh shader based implementation of Marching Cubes, some concepts need to be defined to give a logical structure to the explanation of the algorithm. Section 3.1 describes space segmentation, which outlines how the rendered volume that is evaluated in the Marching Cubes algorithm is segmented. The reason the rendered volume is segmented is to be able to relate it to the structure that the mesh shader pipeline provides. Section 3.2 (Interlinking With Mesh Shaders) describes the relationship between the mesh shader stages and the levels of space segmentation. With the basic form of the implementation defined, we then continue in the following section to propose a set of culling methods to reduce workload. Finally, the implementation of the program is described and specifics of the implementation are decided and justified.

3.1 Space Segmentation

In order to integrate the Marching Cubes algorithm with the mesh shader pipeline, the rendered volume must have an associated logical hierarchy of different scales. This is done to associate different scales of work with the stages in the mesh shader pipeline and provide the means of applying new techniques to compute over the rendered volume.

Figure 3.1 shows the key levels of space segmentation. We describe the overall volume over which the algorithm is being applied as the "total space" (dark grey). The total space is then divided into a regular grid, with a given unit in the grid being named a "subspace". A single example of a subspace is shown in Figure 3.1, highlighted in blue. Each subspace is subdivided into a set of "local spaces" (red). Finally, each local space is divided once more into a set of cells which Marching Cubes uses.

For the sake of explanation, the relative size of each of these scales of space are represented arbitrarily. In practice, they must be defined in the

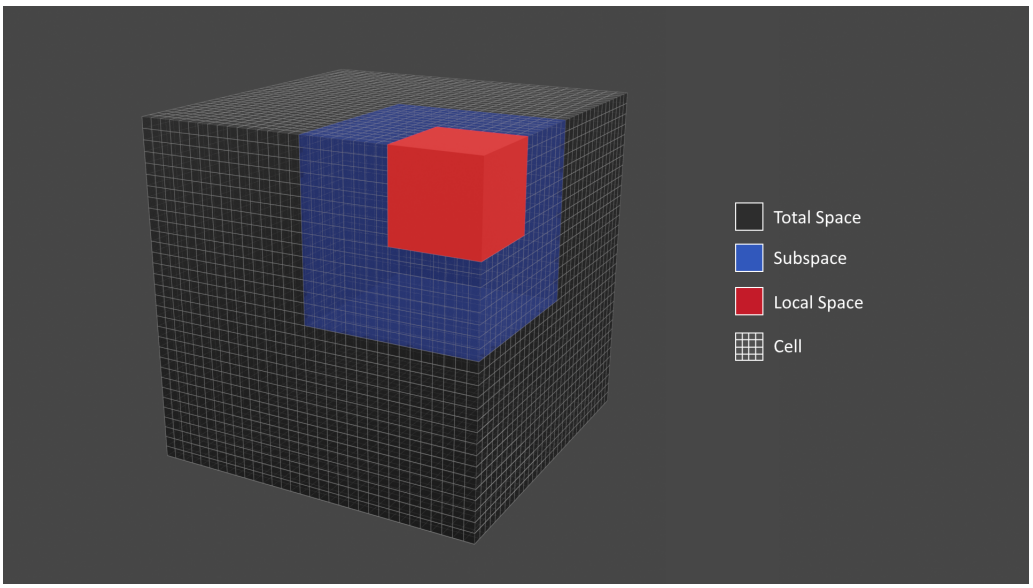


Figure 3.1: Illustration of space segmentation hierarchy.

implementation according to hardware limitations. Observing Figure 3.1, it shows the total space (dark grey), represented as a $32^3 = 32768$ space of cells. It is split into subspaces (blue) of $16^3 = 4096$ cells, making the total space be made up of $2^3 = 8$ subspaces. Each local space (red) is made up of $8^3 = 512$ cells, meaning a subspace is made up of $2^3 = 8$ local spaces.

When creating the implementation, the relative sizing between total area, subspace and local space is dependent on limitations imposed on the mesh shader implementation by the GPU itself and characteristics of the GPU's hardware for optimal execution. For instance, matching the number of cells in a local space to the number of threads in a mesh shader warp typically leads to best use of execution units. The chosen sizes will be outlined in more detail and justified by Section 3.4.

3.2 Interlinking with Mesh Shaders

The shader stages we have control over are: the task shader and mesh shader stages. We also control the number of task shader workgroups to be dispatched through instructions sent by the CPU. Different responsibilities can be taken by each of the shader stages shown in Figure 3.2.

To render the total space, a set of task shader workgroups are dispatched by the CPU. Each task shader workgroup is responsible for a subspace. Each local space making up a subspace will be evaluated in parallel by a thread in the task shader workgroup. Once each local space is processed by the task shader stage, a mesh shader workgroup is launched for each of the local spaces. Finally, each cell in the local space is computed by a thread in the mesh shader

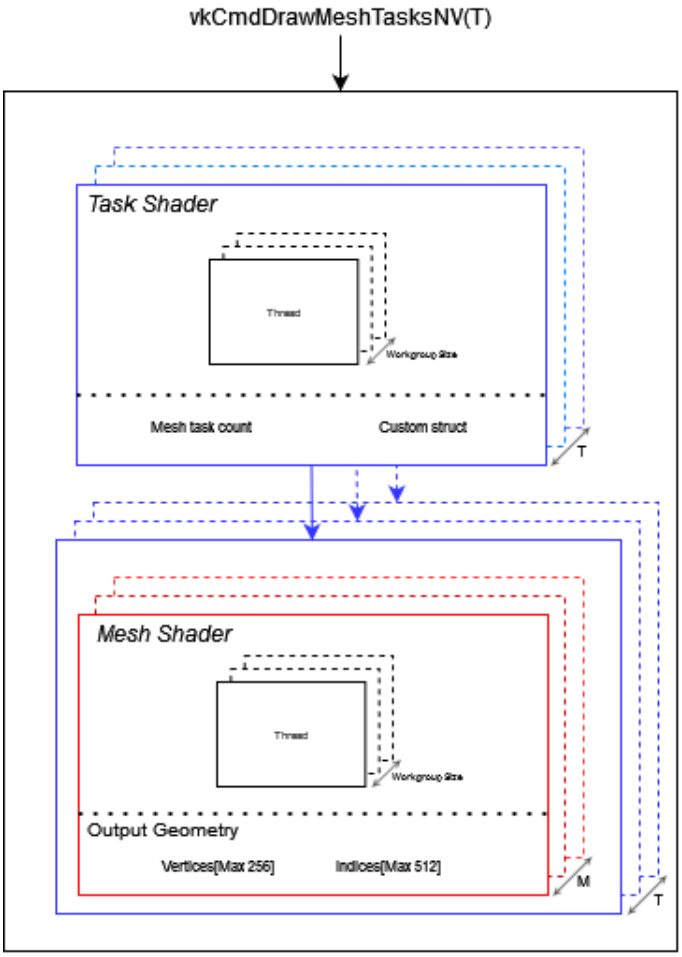


Figure 3.2: An overview of the mesh shader pipeline, showing the relationship between task and mesh shader.

workgroup. It is in the mesh shader that the core decision about the geometry for a particular cell is made and written to the mesh shader output. It is then passed on to the rasterizer for the rest of the stages in the graphics pipeline to be carried out.

3.3 Culling Methods

One of the use cases for task shaders is to reduce unnecessary workload. A brute-force approach would be to launch mesh shader workgroups to evaluate every local space. However, this would be wasteful if most of the rendered volume had no geometry, suggesting one way it might be possible to reduce that workload. Because task shaders control the number of mesh workgroups to launch, if it were possible to determine that no part of the geometry passed through the cells in a local space, then there would be no need to dispatch a mesh shader workgroup for that local space.

When finding large volumes which can be determined to be either inside or

Figure 3.3: 2D Marching Squares example with red highlighted regions noting general areas which could be culled.

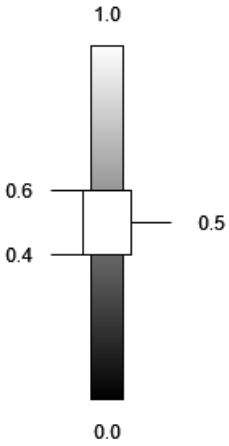


Figure 3.4: Visualisation of tolerance.

outside of the isosurface, meaning they have no geometry, it is not possible to determine this analytically in practice, unless each cell is computed anyway, and so some heuristic is needed. We have devised two methods of evaluating whether a local space should be culled or not. The first approach looks at the centre of the local space, the other looks at each corner. Figure 3.3 shows an example of Marching Squares with general areas highlighted in red which could potentially be left out of evaluation in 3D Marching Cubes. In Marching Cubes, geometry will be constructed at the boundary between inside and outside of the highly dense areas.

As previously noted, these are methods are heuristics and do not provide a guarantee that there is no geometry within a local space. Because of this, a tolerance is applied in each of the methods. With density functions producing values between $0.0 < f(x) < 1.0$, a Marching Cubes threshold of 0.5 and a

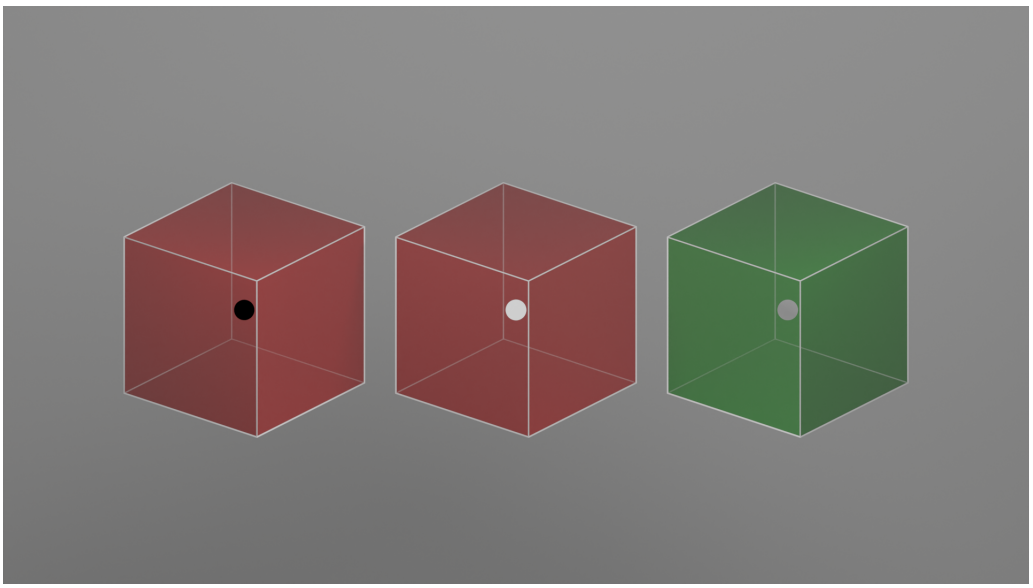


Figure 3.5: Centre culling examples with skipped (red) local spaces and a processed (green) local space.

tolerance of 0.1, a local space would be considered for culling if $|f(x) - 0.5| > 0.1$.

Figure 3.4 shows this, with a value of 0.0 being black and 1.0 being white. Any value which resides in the box in the centre is within the tolerance and its local space would be processed for geometry discovery.

3.3.1 Centre based local space culling

The density function is sampled at the centre of each local space. The value of the sample is then checked for whether it is close to the Marching Cubes threshold. If the value is found to be too far enough away from the density threshold, the local space is assumed to have no geometry and consequently is culled. Figure 3.5 shows local spaces with values far below the threshold (shown in black) and far above (shown in white) which are to be skipped from rendering, in addition to one with a value near the threshold (grey) which is chosen for geometry evaluation.

3.3.2 Corner based local space culling

Each corner of the local space is sampled using the density function. The corners are then checked to see whether they are all inside or all outside of the isosurface, or if there is a combination of inside and outside corners. If all corners are inside or all outside of the highly dense space, the local space is assumed to have no geometry and consequently is culled. Examples of this are

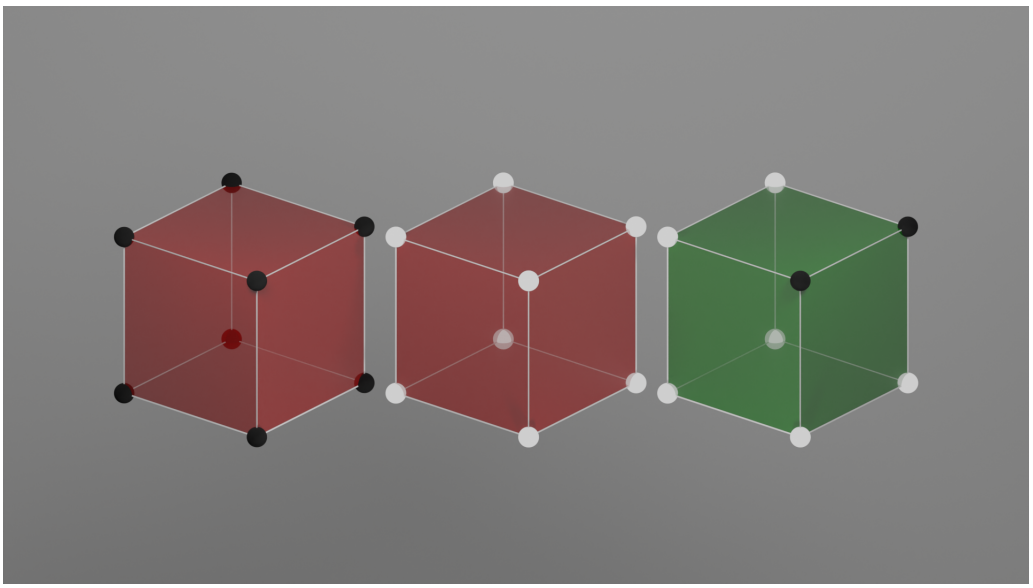


Figure 3.6: Corner culling examples with skipped (red) local spaces and a drawn (green) local space.

shown in Figure 3.6, showing all of the corners being outside of the isosurface (left), inside (centre) and partially inside (right).

3.4 Implementation

We now turn our attention to implementation details of the devised Marching Cubes that utilises the NVIDIA mesh shader pipeline, applied to a space filled with simplex noise. First we provide an overview that describes which language and APIs were used to create the implementation. Then, a description on how subgroup operations are used so as to have a mesh and task shader workgroup work cooperatively. Finally, a description of how the task and mesh shaders are used to create the mesh is given. The full code is available in a GitHub repository.[1]

3.4.1 Overview

The programming languages and graphics API chosen were C++, GLSL and Vulkan. C++ is used to create the CPU program which manipulates the GPU through the Vulkan API. GLSL is used for the shader code. As Vulkan requires shaders in SPIRV bytecode, which is then compiled during graphics pipeline creation by the driver. We used the shader compiler *libshaderc*, to convert the written GLSL shader source code to SPIRV bytecode.

[1] https://github.com/JibbityJobbity/MeshShadersMarchingCubes

The reason for these choices was due to the author's familiarity with the programming development environment and for easier portability between graphics APIs and operating systems. Mesh shaders are supported in OpenGL, which uses shaders written in GLSL and eliminates the requirement for *libshaderc*. However, the Vulkan API provides a lower level control and is much more explicit, thus reducing side effects from GPU drivers. While development was undertaken on Windows, the aforementioned APIs and compiler tools have been ported to all the main operating systems, including Linux and MacOS. A mesh shader implementation based on Direct3D 12 would require HLSL shader code, however as *libshaderc* also supports this shader language, the necessary byte code could be accomplished in a similar way. While efforts have been made for running Direct3D 12 applications on Linux through translation layers to Vulkan, support is limited, and the situation for MacOS is even more underdeveloped.

Sourced from a general tutorial site about Vulkan, a suitable sample program was located, and used as the main scaffolding for this project. The sample program takes the form of a "monolithic" simple rendering application. For the purposes of the development work, and evaluation experiments undertaken, this was sufficient for our needs.[2]

The two primary inputs to the developed Marching Cubes program were: noise scale and subdivisions. Noise scale controls the scale that noise is sampled across the total space. This is so noise scale remains independent from subdivisions. Subdivisions controls the resolution which the Marching Cubes algorithm runs at, effectively giving the ability to increase and decrease workload. The number of subdivisions is constrained to be a multiple of the number of cells a subspace has in one dimension. For instance, if a local space had $2^3 = 8$ cells and a subspace had $4^3 = 64$ local spaces, subdivisions must be a multiple of $2 * 4 = 8$.

3.4.2 Subgroup Operations

When populating meshlet data and identifying which local spaces to draw, subgroup operations are useful for communication within a warp to cooperatively fill buffers. In the Vulkan API, "subgroup" refers to a group of threads that run in parallel (Henning, 2018). These groups can be directly mapped to the concept of a warp on NVIDIA hardware, where they are groups of 32 threads.

Subgroup operators allow for sharing of data within threads of a subgroup. They work similarly to GPU level atomic variables with the difference being

[2] `https://github.com/Overv/VulkanTutorial/blob/master/code/23_descriptor_sets.cpp` Accessed on: 16/10/2021

that they are localised within a subgroup. These operations are implemented in hardware and have less overhead than atomic variables which are synchronised across every thread in the GPU. By using these operations to update counters, output buffers of each task shader or mesh shader workgroup can be filled serially.

3.4.3 Task Shader

In the implementation, a number of task shader workgroups are first launched using the Vulkan API as a command in a command buffer. A task shader workgroup is responsible for processing a single subspace, comprised of $4^3 = 64$ local spaces. The task shader identifies the subspace and local space each thread is processing. It then computes the value at each corner of each local space using a density function—in our case simplex noise. For this step, we took the Java code by Gustavson (2005) that implemented an improved version of simplex noise, and ported it to GLSL. In our GPU-based implementation, each of the density values computed for each corner point is stored in shared memory to prevent redundant computation of the corners of each local space.

Centre culling is done by using the density function to sample the centre of the local space. It is then checked for whether it is close to the threshold. This decision is made on the fly during shader runtime.

Corner culling uses the density function at each corner. In this version of the implementation, both the maximum and minimum value of each of the corners is found. If both the maximum and minimum numbers are either above or below the threshold, it is marked for culling.

Regardless of the culling strategy, the workgroup then cooperatively applies the culling technique chosen on each local space. Local spaces that are *not* to be culled are written to the task shader workgroup's output buffer and one is added to the current count of mesh shader workgroups to launch. This operation is done cooperatively using subgroup ballot operations, allowing each thread in a workgroup to indicate whether its currently processed local spaces are being rendered. For mesh workgroups to identify which meshlets to compute, because the buffer is accessible from mesh workgroups launched by that task workgroup, the task workgroup's custom structure previously shown in Figure 3.2 can be used.

3.4.4 Mesh Shader

With mesh shader workgroups being launched by the task shader stage, it is now the mesh shader's responsibility to evaluate local spaces and produce geometry. In the implementation, a local space is composed of 8, or rather

2^3 marching cube cells. Using the task shader workgroup's output buffer, the local spaces to draw can be identified and their position computed. The mesh shader workgroup then cooperatively iterates over each of the $3^3 = 27$ corners of the Marching Cubes cells, and uses the same density function as the task shader to compute the density of each corner.

The workgroup finally cooperatively iterates over each marching cube cell. This is where the decision for the geometry is made in each cell, previously mentioned in Chapter 2, using the configurations from Figure 2.5. Each cell is examined by a thread to determine whether each corner is inside or outside of the surface. Each corner is stored as a bit in a single integer, which is used as an index into the look-up table of triangle configurations. The edge vertices that are determined to be part of the boundary are added to the list of vertices in the meshlet.

Moreover, size limitations in a meshlet are a concern and are the driving factor for deciding the amount of computation done in a mesh workgroup/local space. The maximum possible number of vertices allowed by GPU hardware in a meshlet is 256, which is a concern when considering the worst case scenario for a local space. Given that the maximum number of triangles in a single Marching Cubes cell is 5, each triangle having 3 vertices; a local space composed of $2^3 = 8$ cubes gives the maximum fit of $2^3 * 5 * 3 = 120$ total vertices. A local space size of $3^3 = 27$ requires space for $3^3 * 5 * 3 = 405$ total vertices and so exceeds the maximum number of vertices in a meshlet. A local space size of $2^3 = 8$ was chosen to guarantee the ability to store every vertex needed.

There are some important details to discuss about this implementation that may lead to inefficiencies. Specifically, in the implementation of the mesh shader. Because of the choice to render 2^3 Marching Cubes cells for a mesh workgroup, there will be 27 points to calculate with the density function and 8 cells for the 32 threads in a workgroup. This causes the effective capacity of each workgroup to be $27/32 = 0.84375$ of a warp's capacity during noise computation and $8/32 = 0.25$ in the Marching Cubes stage, reducing the mesh shader's efficiency to a quarter of a warp's capacity during the cell evaluation stage. Within that, different cells in a local space may have varying numbers of triangles, causing more wastage of threads. In the worst case, if only one cell in a local space had one triangle and the rest had none, the branch mechanism in the cooperative for-loop to process each cell would result in only one thread in the entire workgroup having any meaningful work. A different scheme for determining geometry for each cell might help for performance in local spaces with less geometry.

The choice of symmetric cubic spaces was the result of earlier trials, which were exploring options for vertex sharing between triangles generated by a

mesh shader and gaining efficient occupancy in workgroups. Culling was pursued as an alternative method of gaining performance improvements.

Chapter 4

Evaluation and Discussion

With the algorithm implemented using the mesh shader pipeline, it can be evaluated and compared to the compute shader based implementation by Tlatlik. In addition to a comparison with our baseline mesh shader implementation, the two devised culling methods were also evaluated to determine their effect. Finally, we use performance profiling tools to help create a hypothesis around the current bottleneck and prove it in further performance tests.

4.1 Testing Setup

Testing was run on a machine equipped with an NVIDIA GeForce RTX 2080 graphics card with an Intel i5-6600 processor and 16 GB of memory. The operating system the tests were run on was Windows 10 with an NVIDIA graphics driver version 512.15.

4.2 Performance Measurement Methodology

Measuring the framerate will be used as the performance measurement as it is a good indicator for speed. When measuring for performance, the implementation's viewport camera settings were set uniformally and did not change throughout the running of the tests. Noise randomization was set to use the same random number generator in each test run. With a given seed value, the noise function always generates the same value at any given location regardless of the order and frequency of evaluation calls, so fixing the seed makes sure that the density patterns are the same for each test. This ensures a similar rendered image for each test and reduces variance in rasterizer and fragment workload. Framerate was measured by counting the number of rendered frames and dividing it by the elapsed time in seconds. The elapsed time was taken by measuring the time directly before and after the render loop was executed, terminating the application when the elapsed time exceeded 60 seconds

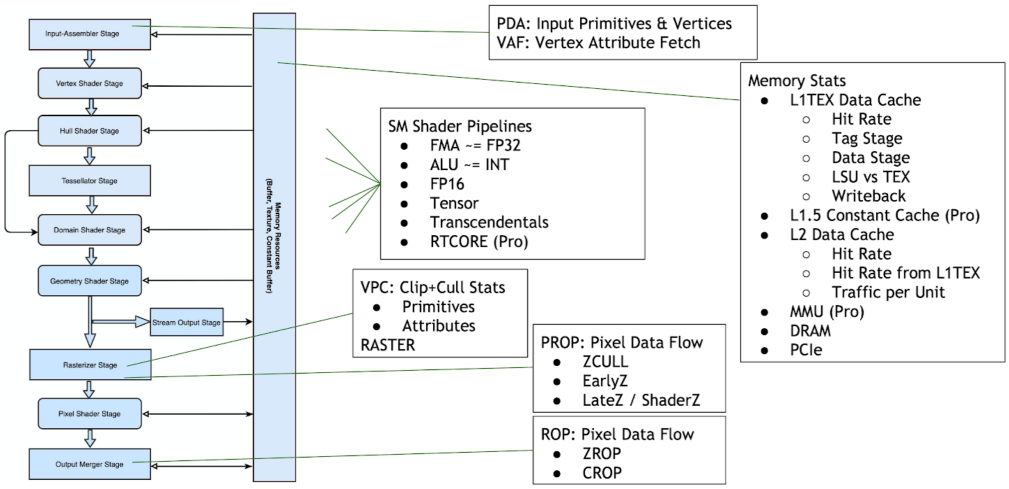


Figure 4.1: Overview of the units which Nsight provides performance measurements. Reproduced from NVIDIA Corporation (2022).

To measure the usage of units inside of the GPU, NVIDIA Nsight Graphics version 2022.01 was used in GPU Profile mode. This is used by graphics programmers to obtain performance metrics inside the GPU to find potential bottlenecks. Figure 4.1 shows an overview of the metrics gathered by Nsight Graphics. In our case, we can use it to identify potential reasons for the way this implementation and the compute shader implementation performs the way they do and suggest possible ways to give improvements, or other types of workloads to interleave.

4.3 Comparison Against the Compute Shader Solution

To properly evaluate whether the new mesh shader solution was viable, a performance comparison of the new application against Tlatlik's compute shader program was made. This was done over a set of different noise scales comparing how each of them performed using each implementation.

4.3.1 Method

We made the performance comparison using simplex noise and measured performance with different noise scales, creating both highly variable and erratic volumes as well as smooth surfaces. The number of computed cells for Marching Cubes remained the same, as the goal was to see whether this approach was faster. From the results, we can draw conclusions as to whether replacing a compute shader solution with the new mesh shader solution is reasonable.

Noise Scale	Resolution	Framerate
64	48	114.2
32	48	118.4
16	48	121.4
8	48	127

Table 4.1: Performance of different noise scales using the compute shader solution.

Noise Scale	Resolution	Framerate
64	48	2679.44
32	48	2677.38
16	48	3041.39
8	48	3688.65

Table 4.2: Performance of different noise scales using the mesh shader solution.

4.3.2 Results and Evaluation

Table 4.1 shows the performance for the compute shader solution and Table 4.2 shows performance for the new mesh shader solution. Noise scale is the relative scale of the noise functions to a total space. Resolution is the number of marching cube cells in each dimension, increasing the computational load. As shown by the results, the mesh shader solution proves to be much faster for a range of workloads. In large noise scales giving erratic volumes, the framerate is lower across both solutions. However, we see increases from 114.2 frames per second (fps) in the compute shader implementation, to 2679.4 fps in the mesh shader implementation. Similarly, in a smaller and smoother noise scale, the framerate increases from 127.0 fps to 3688.65 fps.

When profiling the compute shader implementation using NVIDIA Nsight, Figure 4.2 showed there was a bottleneck related to VRAM usage. VRAM (Video Random Access Memory) is the main memory of the GPU, shared by all executing threads in all SM's. In particular, VRAM write showed itself as a bottleneck in Figure 4.3. This in turn was because noise data was generated in a separate compute shader stage, written to GPU memory and subsequently read during the Marching Cubes stage. The resulting mesh was then stored and subsequently read in the graphics pipeline.

The mesh shader approach bypasses the need for writing or reading re-

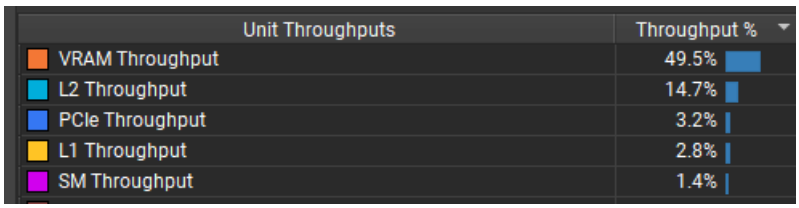


Figure 4.2: Unit throughputs while running the compute shader solution.

Figure 4.3: VRAM bandwidth while running the compute shader solution.

sources to/from VRAM as it only requires resources within shared memory, localised inside each SM. This alleviates the bottleneck caused by VRAM. This allows generation of highly dynamic volumes due to the nature of the mesh shader implementation. Everything is generated inline without any dependence on VRAM access, from the input data for Marching Cubes to the resulting mesh.

Figure 4.4 shows the usage of each unit within the GPU for the mesh shader solution. The most usage was to do with SM throughput and PES+VPC throughput (viewport clipping and culling). However, none of these units are close to being fully utilised. Throughput for memory access remains low, whether it be through cache, VRAM or CPU system memory. This indicates that the performance inhibitor for this implementation is more likely to do with SM occupancy and/or latency bound. This indicates that performance is available to read predefined input data in medical contexts. This would be interesting to investigate in future work.

Overall, this experiment shows a significant improvement in the case of generated input data using noise functions and dynamically created input data. Because the mesh is evaluated during the time of the draw instruction, there would be minimal impact when generating volumes in new positions in each frame, allowing moving through the noise or changing the Marching Cubes threshold in realtime.

4.4 Culling Modes

The usefulness of the novel culling methods for local spaces needs to be determined. To establish usefulness of the two different culling methods, different resolutions of Marching Cubes were run with different noise scales and toler-

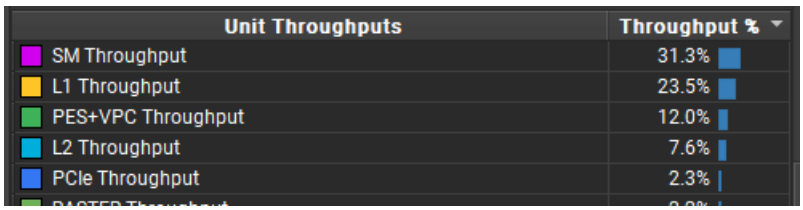


Figure 4.4: Unit throughputs while running the mesh shader solution.

ances in each method. The framerate as well as accuracy were measured to evaluate effectiveness.

4.4.1 Method

To create a method of comparing culling strategies, we needed to measure performance and the actual accuracy of each method. In this way, we can know the magnitude of gain or loss in performance was well as rating their rendering accuracy.

The effectiveness of culling methods was determined by recording whether each local space was chosen for culling. This was done by writing modified versions of the task and mesh shaders which determined if a local space would be chosen for culling, but processed it anyway. Whether each local space was chosen for culling and the number of primitives actually drawn was written into a new buffer in GPU memory and that buffer was stored in a file. Performance measurements and accuracy measurements were done independently as measuring accuracy had a significant effect on performance due to writing and copying GPU memory to the CPU for processing. Performance measurements were made on the unmodified program displaying the same geometry.

When evaluating a local space, a result was deemed positive when it needed drawing and negative for it being possible to cull. Whether the result was true is dependent on whether it was elected for rendering and it had geometry, or if it was culled and there was no geometry. True Positives (TP) are when the meshlet was correctly determined to have geometry. In the contrary condition, False Negative (FN) results are ones that were culled but had geometry.

Three metrics were calculated from the raw data gathered: precision, recall and Missed Geometry %. Precision is calculated by

$$Precision = \#TruePositive/(\#TruePositive + \#FalsePositive)$$

and is ideal when it is closest to 1.0. It represents the fraction of work that went into rendering local spaces that actually had geometry rather than being wasted on empty space.

Recall is calculated by:

$$Recall = \#TruePositive/(\#TruePositive + \#FalseNegative)$$

It represents the amount of local spaces that were not culled out of the total number of local spaces with geometry. Culled Work shows the percentage of the total work which was culled and provides insight into how much computation was culled overall.

Missed Geometry % shows the percentage of geometry that was culled by the heuristic being evaluated, when it should not have been. This comes from local spaces which were chosen for culling but actually had meaningful work to be done, causing loss of geometry (which may or may not be visible to the user). Having this metric be strictly 0 and recall be 1 would be ideal as any values which differ result in missed geometry.

4.4.2 Results

A series of tests were run with a set of different Marching Cubes resolutions, culling strategies and varying tolerance settings. Tables 4.3–4.6 show the measured results and derived figures. The scale of simplex noise is set to a scale of 2, giving a consistent volume to compute at different resolutions. The columns labelled TP, FP, FN and TN are abbreviations of true positive, false positive, false negative and true negative respectively. In corner culling, the terms "exclusive" and "inclusive" are used. These tests run with a tolerance setting of 0 with the comparison for whether each point was above or below the threshold being altered between exclusive and inclusive.

Table 4.3 shows the results for varying tolerances when setting the implementation to run at a Marching Cubes resolution of $256^3 = 16,777,216$ cells and using the corner culling technique. Table 4.3a shows the accuracy measurements recorded and Table 4.3b shows the computed figures which represent the rated accuracy and performance and the final framerate. Table 4.4 shows the same results for corner culling. Tables 4.4–4.6 show the two culling methods for a Marching Cubes resolution of $128^3 = 2,097,152$. Note that the range of tolerances is different between corner and centre culling experiments. This is to clearly show the tolerance value at which false negatives start to appear.

4.4.3 Evaluation

There is a clear performance boost as each culling strategy is tuned to be more aggressive. The gathered data shows the corner culling method is able to reach higher precision before losing recall. This means that it could cull

Tolerance	TP	FP	FN	TN	TP Tri #	FN Tri #
Exclusive	25249268	0	226606	396051678	203012223	863085
Inclusive	25472909	225398	2965	395826280	203869138	6170
0.000015	25473949	676512	1925	395375166	203871270	4038
0.000225	25475874	1606070	0	394445608	203875308	0
0.025	25475874	103762410	0	292289268	203875308	0
Off	25475874	396051678	0	0	203875308	0

(a) Measurements

Tolerance	Precision	Recall	Missed Geometry	Culled Work	Framerate
Exclusive	1.0000	0.9911	0.4233%	94.01%	595.2730
Inclusive	0.9912	0.9999	0.0030%	93.90%	588.3160
0.000015	0.9741	0.9999	0.0020%	93.80%	587.2220
0.000225	0.9407	1.0000	0.0000%	93.58%	572.7770
0.025	0.1971	1.0000	0.0000%	69.34%	164.8030
Off	0.0604	1.0000	0.0000%	0.00%	65.5886

(b) Computed accuracy

Table 4.3: Results of corner culling at 256^3 resolution.

Tolerance	TP	FP	FN	TN	TP Tri #	FN Tri #
0.005	18323250	3096000	7152624	392955678	175983150	27892158
0.01	24302830	18194936	1173044	377856742	200141093	3734215
0.02	25475874	58872192	0	337179486	203875308	0
0.03	25475874	98887215	0	297164463	203875308	0
0.05	25475874	170845961	0	225205717	203875308	0
0.1	25475874	296607308	0	99444370	203875308	0
Off	25475874	396051678	0	0	203875308	0

(a) Measurements

Tolerance	Precision	Recall	Missed Geometry	Culled Work	Framerate
0.005	0.8555	0.7192	13.6810%	94.92%	704.8820
0.01	0.5719	0.9540	1.8316%	89.92%	404.8330
0.02	0.3020	1.0000	0.0000%	79.99%	230.8980
0.03	0.2049	1.0000	0.0000%	70.50%	170.1910
0.05	0.1298	1.0000	0.0000%	53.43%	120.4050
0.1	0.0791	1.0000	0.0000%	23.59%	81.8434
Off	0.0604	1.0000	0.0000%	0.00%	66.0094

(b) Computed accuracy

Table 4.4: Results of centre culling at 256^3 resolution.

Tolerance	TP	FP	FN	TN	TP Tri #	FN Tri #
Exclusive	6302793	0	58572	46329579	50764430	212282
Inclusive	6358474	56447	2891	46273132	50970402	6310
0.000015	6358891	160623	2474	46168956	50971276	5436
0.000225	6360791	291066	574	46038513	50975510	1202
0.025	6361365	12788847	0	33540732	50976712	0
Off	6361365	46329579	0	0	50976712	0

(a) Measurements

Tolerance	Precision	Recall	Missed Geometry	Culled Work	Framerate
Exclusive	1.0000	0.9908	0.4164%	88.04%	2054.3100
Inclusive	0.9912	0.9995	0.0124%	87.83%	2010.1700
0.000015	0.9754	0.9996	0.0107%	87.63%	2005.0400
0.000225	0.9562	0.9999	0.0024%	87.38%	1983.2500
0.025	0.3322	1.0000	0.0000%	63.66%	924.3690
Off	0.1207	1.0000	0.0000%	0.00%	444.6470

(b) Computed accuracy

Table 4.5: Results of corner culling at $128^3 = 2,097,152$ resolution.

Tolerance	TP	FP	FN	TN	TP Tri #	FN Tri #
0.005	2667696	55246	3693669	46274333	31039192	19937520
0.01	4576647	753677	1784718	45575902	44051028	6925684
0.02	6062219	4464055	299146	41865524	50040407	936305
0.03	6343772	9220429	17593	37109150	50926641	50071
0.05	6361365	18172401	0	28157178	50976712	0
0.1	6361365	33890270	0	12439309	50976712	0
Off	6361365	46329579	0	0	50976712	0

(a) Measurements

Tolerance	Precision	Recall	Missed Geometry	Culled Work	Framerate
0.005	0.9797	0.4194	39.1110%	94.83%	3671.5500
0.01	0.8586	0.7194	13.5860%	89.88%	2403.3400
0.02	0.5759	0.9530	1.8367%	80.02%	1460.9200
0.03	0.4076	0.9972	0.0982%	70.46%	1094.1800
0.05	0.2593	1.0000	0.0000%	53.44%	783.9770
0.1	0.1580	1.0000	0.0000%	23.61%	545.3020
Off	0.1207	1.0000	0.0000%	0.00%	167.3680

(b) Computed accuracy

Table 4.6: Results of centre culling at $128^3 = 2,097,152$ resolution.

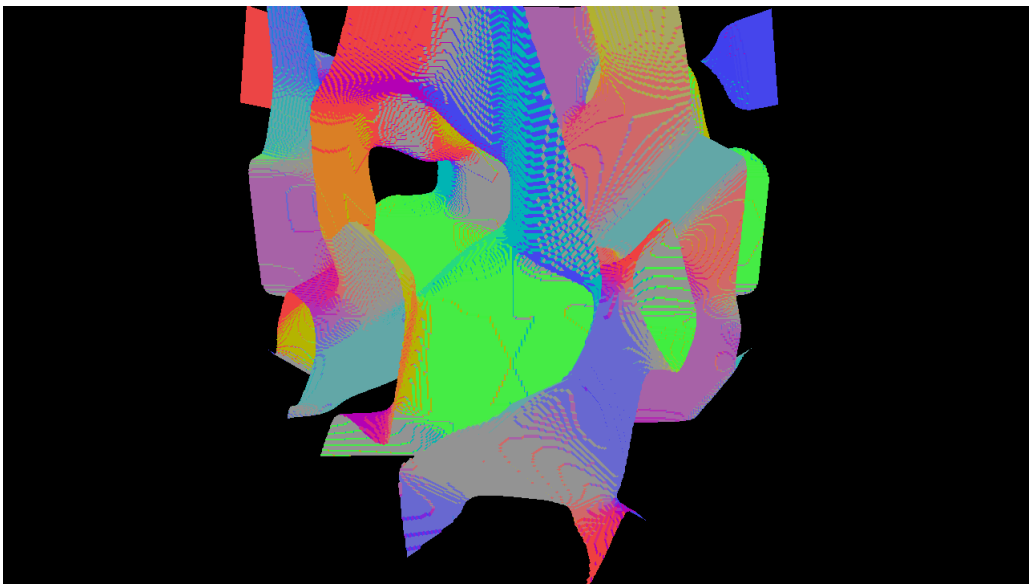


Figure 4.5: Marching Cubes algorithm running correctly with no culling artefacts.

more empty local spaces before starting to (incorrectly) cull local spaces which had geometry. Therefore, corner culling is preferred over centre culling because it safely culls more work, and thus producing a higher framerate.

Corner culling in Table 4.3 demonstrates its ability to retain precision up to 0.9407 before beginning to lose recall. In tolerance setting 0.000015, precision increases to 0.9741 when recall goes down to 0.9999. Note that any recall value which is not 1.0 means there is missing geometry.

When comparing this to centre culling with a tolerance setting of 0.1 shown in Table 4.4, precision only goes to 0.5719 before recall drops to 0.9540. This has an effect on the framerate, where corner culling at a tolerance of 0.000225 has a framerate of 572.7770 fps and centre culling, where at 0.2, being the most conservative tolerance setting before losing recall, the framerate is 230.8980 fps. Corner culling shows the ability to reach higher precision before losing recall, making it more effective at culling work and providing a better framerate.

This is not to say that conservative tolerances have no use case. An example, such as 0.025 in the $128^3 = 2,097,152$ total space, shows a large increase in framerate from 444.647 without culling to 924.369 with the conservative setting. Even though corner culling at the resolution $256^3 = 16,777,216$ with a tolerance of 0.025 shows a precision of 0.1961, there is a clear increase of framerate from 65.5886 to 164.8030. However, further data does show that decreasing the tolerance, causing a more aggressive culling strategy, has the effect of obtaining higher precision while retaining recall of 1.0 (not missing geometry).

Although culling methods do provide a performance improvement as the

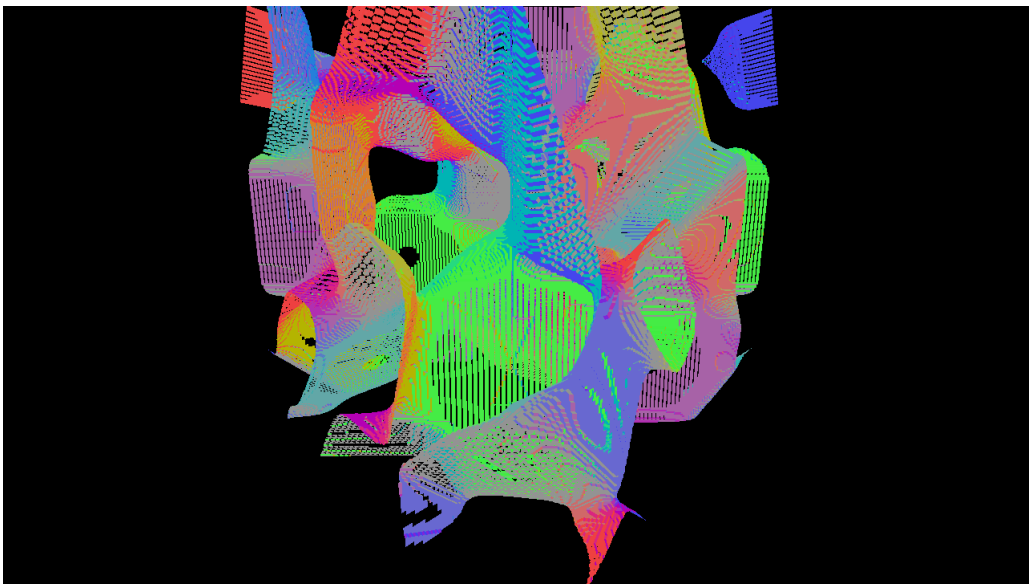


Figure 4.6: Marching Cubes with a culling strategy tuned too aggressively.

tolerances are tuned to be more aggressive, tuning these too aggressively can cause local spaces to be culled when they had meaningful geometry. This causes artefacts in the final mesh which may be visible and undesirable to the user, depending on the use case of the implementation.

Centre culling with a setting of 0.005 on a Marching Cubes resolution of $(32 * 8)^3 = 256^3 = 16,777,216$ is an example with artefacts visible. The displayed output is shown in Figure 4.6. The artefacts are the black stripes and marks on the coloured surfaces—missing geometry lets the black background show through. Stripes and marks of contrasting colour, like red in the central green area where missing geometry lets hidden geometry show through. This is in contrast Figure 4.5, which shows the correct output with culling disabled. The rendered image shown in Figure 4.5 is the result of a volume with $256^3 = 16,777,216$ cells. The colour of each face is given by calculating normal vector for each, then scaling the resulting vector to be shown in visible colour space. The colour banding shown is the combination of flat face shading with sharp edges between triangles. This effect is most visible in the blue section near the top of the middle in the figure. The distinction in Figure 4.6 is that it shows where parts of the mesh are missed, showing the black background or the colour of another surface. This is with corner culling set with a tolerance of 0.1. Here, the profile of the noise moves too far away from the tolerance too quickly at the centre of those local spaces.

The effect that changing Marching Cubes resolution has on each culling strategy is that it makes each culling strategy less effective for the same given tolerance setting. When comparing $256^3 = 16,777,216$ and $128^3 = 2,097,152$ Marching Cubes resolutions, lowering the resolution causes the effect of having

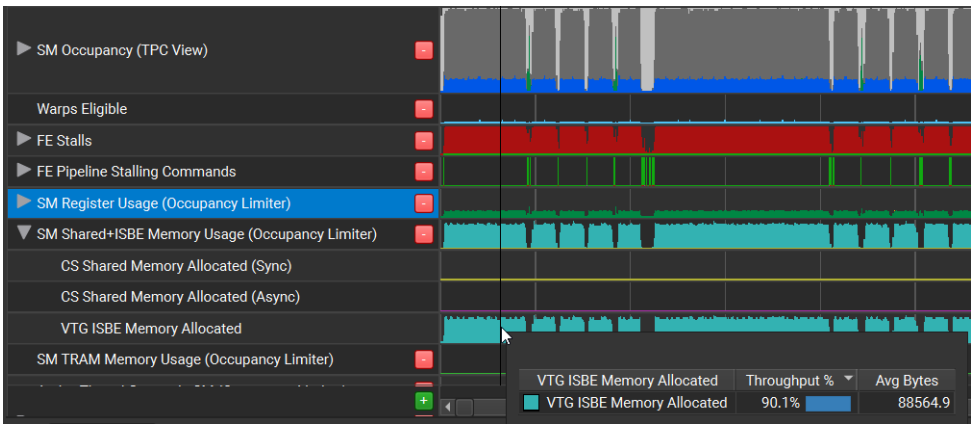


Figure 4.7: Shared memory usage.

each culling strategy be more aggressive, where there is a general increase in precision and decrease in recall given the same tolerance setting. This shows that it is necessary to tune the culling algorithms to be more conservative when using lower resolutions to prevent missed geometry. On the contrary, culling is allowed to be tuned more aggressively when rendering the volume at a finer Marching Cubes resolution relative to the same profile of noise. This makes tuning each of the culling strategies more aggressively more reasonable as Marching Cubes resolution increases.

4.5 Performance Investigation

Gaining insight into the reason the mesh shader solution performs the way it does when the problem scales up can help with potential ideas for optimisation. Using the same performance measurement tools as from earlier, we were able to identify what limits performance.

4.5.1 Method

In order to be able to view more performance metrics, temporarily changing the GPU from an NVIDIA GeForce RTX 2080 up to an RTX 3080 was required. This temporary change in architecture from Turing to Ampere was not for gaining performance, but rather for the profiler feature set which provided new hardware utilisation metrics. Using this extended set of metrics, a hypothesis for what could be limiting performance may be tested on the Turing architecture and can be generally applied to more GPUs.

Profiling on the RTX 3080 showed a significant usage in ISBE Memory Allocated, as seen in Figure 4.7. This shows average usage of shared memory/L1 cache across every SM. In the traditional graphics pipeline, it is used to store

vertex attributes such as normal vectors and texture co-ordinates in both the output of the vertex shader and input of the fragment shader. In the mesh shader context, it is where the output buffer for the meshlet is stored. Figure 4.7 gives a clear indication that shared memory is exhausted, not leaving enough space for any more mesh workgroups. For mesh shader warps to be actively executing on an SM, there must be enough shared memory available to store the output buffer, so the lack of shared memory means that fewer mesh warps are being scheduled onto the GPU, which leads to the rest of the GPU being underutilised. Because of this, we can determine the performance limiter to be the lack of space in shared memory.

To establish this was a fundamental bottleneck due to the implementation we returned to the RTX 2080 GPU to run some additional performance tests. This time, we focused on increasing the memory requirements in the mesh shaders, with the expectation that this would have a negative impact on framerate. The methodology for testing was similar to the previous tests where the average framerate was measured over a set period of time of 60 seconds.

The memory required by a mesh shader warp is the buffer used to store its output geometry. It must be large enough to hold the largest amount of geometry that a mesh shader warp could generate. It is set in the code as the maximum number of vertices and indices. Varying this maximum number between smaller and more wasteful allocations, and measuring the framerate for each configuration, should make the limitation in shared memory more apparent. To retain consistency between tests, buffer overflow guards in the mesh shader were disabled. This ensured that the same amount of geometry was being processed. In practice, this is not recommended as limiting the amount of memory a meshlet has can cause mesh workgroups to write into other buffers that do not belong to them and results in unpredictable artefacts in the rendered output.

#Max Primitives was left unchanged at 512 as each index was an 8-bit number, so that the memory allocated was small and it might have been difficult to observe effects from changes. Another reason was due to the granularity of memory allocation for the index buffer, as noted by Kubisch (2018).

4.5.2 Results and Evaluation

The results shown in Table 4.7 show a clear increase in framerate as less memory is allocated for each meshlet. When observing the difference in framerate between 256, 128 and 64 vertices, a significant increase in framerate is observed. Furthermore, Nsight shows a higher occupancy of mesh shader warps. This metric is shown in Vertex/Tess/Geometry Warps Throughput % and showed the average occupancy of actively running mesh/task shader warps across all

#Max Vertices	#Max Primitives	Framerate
256	512	468.402
192	512	639.79
160	512	732.235
128	512	786.103
96	512	881.77
64	512	1027.19

Table 4.7: Performance across varying memory allocation settings.

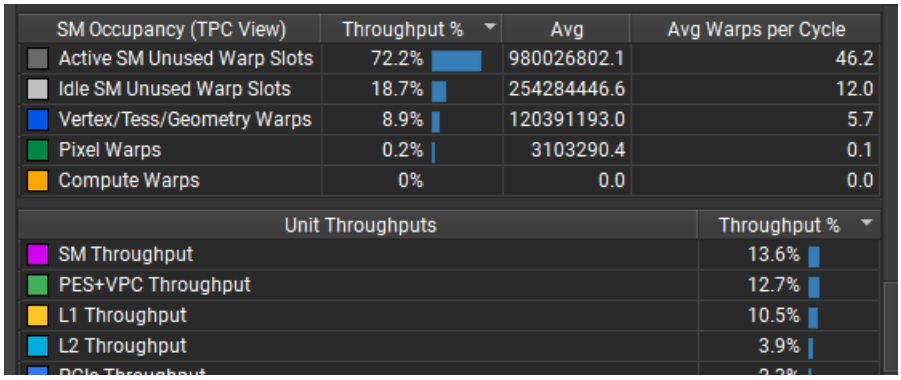
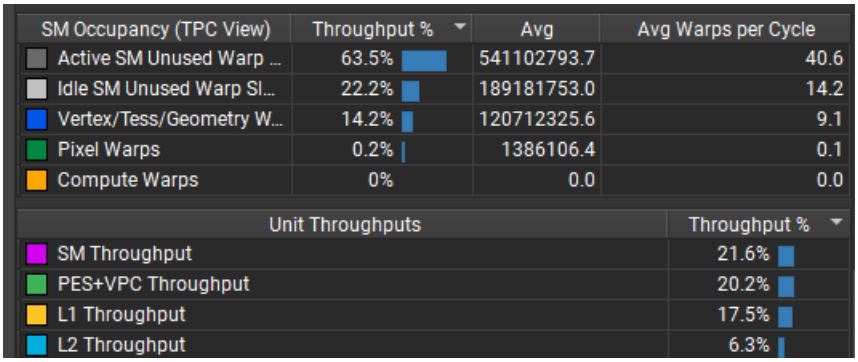


Figure 4.8: SM Occupancy and unit throughputs for the 256 vertices configuration.

SMs over time. Figure 4.8 has a lower occupancy of 8.9% active warp slots in an SM in contrast to Figure 4.9 showing an occupancy of 14.2%, with this effect being further demonstrated in Figure 4.10's 19.9%. Throughputs of the GPU's units show a proportional increase with Vertex/Tess/Geometry Warps Throughput %. The trend of lower memory requirements for meshlets giving better performance strongly suggests that an important factor for performance in mesh shaders is to consider shared memory requirements for workgroups.

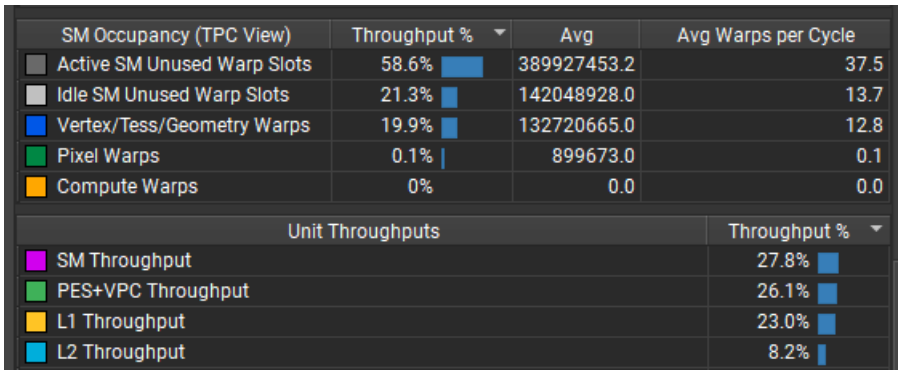
When a workgroup is to be executed, it will require an amount of shared memory. In the mesh shader's case, shared memory is where the meshlet's buffer is stored along with anything else that might be declared as being stored in shared memory. While shared memory has much faster access compared to VRAM, the amount of shared memory on any given SM is significantly more restrictive. Therefore, it is important to consider the size of meshlets and any other shared memory allocation as it is a crucial factor for efficiently occupying an SM.



SM Occupancy (TPC View)	Throughput %	Avg	Avg Warps per Cycle
Active SM Unused Warp ...	63.5%	541102793.7	40.6
Idle SM Unused Warp Sl...	22.2%	189181753.0	14.2
Vertex/Tess/Geometry W...	14.2%	120712325.6	9.1
Pixel Warps	0.2%	1386106.4	0.1
Compute Warps	0%	0.0	0.0

Unit Throughputs	Throughput %
SM Throughput	21.6%
PES+VPC Throughput	20.2%
L1 Throughput	17.5%
L2 Throughput	6.3%

Figure 4.9: SM Occupancy and unit throughputs for the 128 vertices configuration.



SM Occupancy (TPC View)	Throughput %	Avg	Avg Warps per Cycle
Active SM Unused Warp Slots	58.6%	389927453.2	37.5
Idle SM Unused Warp Slots	21.3%	142048928.0	13.7
Vertex/Tess/Geometry Warps	19.9%	132720665.0	12.8
Pixel Warps	0.1%	899673.0	0.1
Compute Warps	0%	0.0	0.0

Unit Throughputs	Throughput %
SM Throughput	27.8%
PES+VPC Throughput	26.1%
L1 Throughput	23.0%
L2 Throughput	8.2%

Figure 4.10: SM Occupancy and unit throughputs for the 64 vertices configuration.

Chapter 5

Conclusion

This thesis documents the development and evaluation of an implementation of the Marching Cubes algorithm using the mesh shader pipeline. Prior work using compute shaders has shown that a GPU-based implementation of Marching Cubes is feasible and faster than a CPU implementation. In this thesis, we have redeveloped the algorithm to take advantage of the new mesh shader pipeline.

To create an implementation of Marching Cubes that fits with the mesh shader pipeline, the problem of the Marching Cubes algorithm was divided to map it to the task and mesh shader stages in the mesh shader pipeline. This was done by segmenting the total rendered space into subspaces and local spaces, handled by the task shader and mesh shader respectively. Using this, local spaces can be culled using the task shader, effectively reducing workload. Two variants of a culling heuristic were created: centre culling and corner culling.

The mesh shader implementation was evaluated in two ways. Firstly, a performance comparison was made against the compute shader solution by Tlatlik. Secondly, the two variants of a culling heuristic were evaluated for efficacy and efficiency.

In conclusion, comparing mesh shader and compute shader solutions for implementing Marching Cubes shows a significant performance improvement in the mesh shader solution. It is worth considering for implementation as a substitute for a compute shader solution. It alleviates the usage of GPU memory (by eliminating dependence on intermediate stages) to store the mesh which leaves space for other resources and does not use video memory bandwidth.

Although performance improvements were gained, the complexity of mesh shaders proved difficult to work with. Particularly, Marching Cubes and proper shading for the geometry. Because of this implementation, there were significant restrictions when it came to implementing smooth normal vectors.

Through this work we have also introduced two culling methods—one centre-based, the other corner-based—that can be used to dramatically reduce the computational cost for rendering, seeing increases from 65.6 to 572.8 frames per second before losing any geometry. The two methods, however, are heuristic rather than analytically derived determinations, and consequently, when pursued too aggressively, can lead to artefacts in the form of missed geometry in the form of holes in the rendered result. In our implementation these manifest mostly as black dots or stripes appearing in the rendered surface. Out of the two culling methods shown, corner culling is more effective at retaining mesh accuracy while culling more work. Examples of this are centre culling achieving a framerate of 230.9 while corner culling achieves 572.8 frames per second before losing geometry.

5.1 Future Work

There are three primary areas in which we propose future work. The first is a comment on the limitations of the current implementation and a proposal for a new one. We then move on to discuss the current culling methods. The final is to handle scanned data from a CT or MR scan rather than using noise functions to populate the input data.

5.1.1 Marching Cubes Implementation

In this implementation, we found an issue trying to use the output meshlet buffer efficiently along with the set of threads in a warp providing meaningful work.

In reviewing the mesh shader implementation of Marching Cubes, a performance analysis was conducted. The results of said performance analysis showed that limited space in L1 cache/shared memory was the primary performance inhibitor. Encoding normal vectors in a smaller data format should give a performance improvement.

The sizes for subspaces and local spaces were kept in symmetric sizes, being 4^3 local spaces and 2^3 cells respectively. These sizes were chosen because they were easy to implement and would be ideal for vertex sharing without consideration for whether they were optimal for the current implementation. Perhaps creating a space segmentation scheme for asymmetric local space sizes such as $2*2*4 = 16$ would result in needing $(2*2*4)*5*3 = 240$, maximising a meshlet's vertex count while having half of a workgroup evaluate cells rather than a quarter in the current implementation. It is not clear whether this would be an improvement, however, as having a larger meshlet has implications on

performance as found in the performance analysis.

An alternative path of action is to create a new space segmentation scheme. A proposed solution would be to evaluate each cell in the task shader stage, encode it in the task shader's output buffer and launch the appropriate number of mesh workgroups to decode the task shader's output and produce geometry. There would have to be considerations over efficiently using task shaders and whether the size of their output buffers would affect performance (Mihut, Kubisch, & Kraemer, 2021).

5.1.2 Culling

The currently given culling methods give a performance improvement when applied. Examples such as Table 4.3 with the tolerance setting at 0.025 improves the framerate from 65.6 to 164.8. Corner culling is much preferred over centre culling as a more conservative tolerance of 0.03 using centre culling, shown in Table 4.6b has a lower recall of 0.9972 when compared to a more aggressive tolerance corner culling, shown in Table 4.6b. Because of the loss of accuracy, it is important to tune the culling tolerance, given the relative scale of noise to Marching Cubes resolution.

Inaccuracies in centre culling come from two areas. The first is where noise stays within the tolerance, but in a curve where is does not cross the threshold. This can cause large amounts of wasted work near the threshold. The centre method can also fail when noise is erratic and moves through the threshold too quickly, causing missed geometry.

The case for corner culling is similar, where noise which crosses the threshold away from the corners is not considered during culling. This could be a cause for missed geometry, although the exact reason for this has not been investigated.

After finding that the preferred method of culling is corner-based local space culling, there is still potential for improvement. Some more elaborate techniques for more accurate culling methods would be interesting to investigate. Enhanced culling could involve more computational requirements and complexity. Although, it could introduce benefits of increased accuracy. A culling technique with complete accuracy and no tuning required would be ideal.

5.1.3 Scanned Data

The Marching Cubes implementation detailed in this thesis uses data generated and immediately used within each draw operation, largely removing dependence on GPU memory access. Applying Marching Cubes to scanned

data, such as the large amounts of data provided by medical scans, presents additional challenges. Consequently, implementing a data-driven version of the algorithm which uses data from an MR or CT scan, as a substitute for noise functions, needs investigation. Considering that the performance limitation in Tlatlik's method was to do with VRAM write throughput, reading the volume data from VRAM may have considerable performance impact. The mesh shader solution developed in this thesis leaves VRAM bandwidth free, and it should be possible to take advantage of this in processing data.

The currently devised culling methods may perform differently with scanned data. For instance, in scanned data, edges between soft tissue and bone would be too distinct for centre culling. In addition, artefacts from culling may be more important when considering a medical context. Therefore, if corner culling proves to be ineffective, alternative culling methods should be considered for scanned medical data.

Finally, a more complete implementation with shading and texturing should be created to evaluate how the mesh shader implementation integrates with a realistic application. Usefulness and complexity for integration has yet to be determined for a medical context or graphical effects using Marching Cubes in video games or other realtime rendering contexts.

References

Blinn, J. F. (1982). A generalization of algebraic surface drawing. *ACM transactions on graphics (TOG)*, *1*(3), 235–256.

Dyken, C., Ziegler, G., Theobalt, C., & Seidel, H.-P. (2008). High-speed marching cubes using histopyramids. In *Computer graphics forum* (Vol. 27, pp. 2028–2039).

Englert, M. (2020). Using mesh shaders for continuous level-of-detail terrain rendering. In *Special interest group on computer graphics and interactive techniques conference talks* (pp. 1–2).

Ghorpade, J., Parande, J., Kulkarni, M., & Bawaskar, A. (2012). Gpgpu processing in cuda architecture. *arXiv preprint arXiv:1202.4347*.

Gong, S., & Newman, T. S. (2013). A corner feature sensitive marching squares. In *2013 proceedings of ieee southeastcon* (pp. 1–6).

Gustavson, S. (2005). Simplex noise demystified. *Linköping University, Linköping, Sweden, Research Report*.

Henning, N. (2018). *Vulkan subgroup tutorial.* `https://www.khronos.org/blog/vulkan-subgroup-tutorial`. (Accessed: 2022-6-16)

Kilgariff, E., Moreton, H., Stam, N., & Bell, B. (2018). *NVIDIA turing architecture in-depth.* `https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/`. (Accessed: 2022-5-6)

Kubisch, C. (2018). *Introduction to turing mesh shaders.* `https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/`. (Accessed: 2022-5-6)

Lorensen, W. E., & Cline, H. E. (1987). Marching cubes: A high resolution 3D surface construction algorithm. *ACM siggraph computer graphics*, *21*(4), 163–169.

Mihut, A., Kubisch, C., & Kraemer, M. (2021). *Advanced api performance: Mesh shaders.* `https://developer.nvidia.com/blog/advanced-api-performance-mesh-shaders/`. (Accessed: 2022-5-6)

Müller, T., Evans, A., Schied, C., & Keller, A. (2022, July). Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, *41*(4), 102:1–102:15. Retrieved from `https://doi.org/10.1145/3528223.3530127` doi: 10.1145/3528223.3530127

Newman, T. S., & Yi, H. (2006). A survey of the marching cubes algorithm. *Computers & Graphics*, *30*(5), 854–879.

NVIDIA Corporation. (2018). *NVIDIA turing gpu architecture* (Tech. Rep.). Retrieved from `https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`

NVIDIA Corporation. (2022). *Nsight graphics.* `https://docs.nvidia.com/nsight-graphics/2022.1/AdvancedLearning/index.html`. (Accessed: 2022-6-17)

Perlin, K. (1985). An image synthesizer. *ACM Siggraph Computer Graphics*, *19*(3), 287–296.

Perlin, K. (2002a). Improving noise. In *Proceedings of the 29th annual conference on computer graphics and interactive techniques* (pp. 681–682).

Perlin, K. (2002b). Noise hardware. In *Real-time shading languages* (pp. 2-1–2-24). CRC Press.

Santerre, B., Abe, M., & Watanabe, T. (2020). Improving gpu real-time wide terrain tessellation using the new mesh shader pipeline. In *2020 nicograph international (nicoint)* (pp. 86–89).

Tlatlik, M. L. (2019). *Volume rendering with marching cubes and async compute.*

Unterguggenberger, J., Kerbl, B., Pernsteiner, J., & Wimmer, M. (2021). Conservative meshlet bounds for robust culling of skinned meshes. In *Computer graphics forum* (Vol. 40, pp. 57–69).

Volkov, V. (2016). *Understanding latency hiding on gpus.* University of California, Berkeley.

Wang, W., Jiang, Z., Qiu, H., & Li, W. (2012). Real-time simulation of fluid scenes by smoothed particle hydrodynamics and marching cubes. *Mathematical Problems in Engineering*, *2012*.

Wang, Y., Zhong, Z., & Hua, J. (2020). Deeporgannet: On-the-fly reconstruction and visualization of 3d / 4d lung models from single-view projections by deep deformation network. *IEEE Transactions on Visualization and Computer Graphics*, *26*(1), 960-970. doi: 10.1109/TVCG.2019.2934369

Xiang, Y., Choi, W., Lin, Y., & Savarese, S. (2015). Data-driven 3d voxel patterns for object category recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1903–1911).