

Universidade do Minho

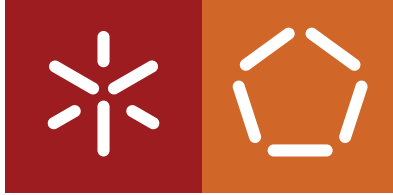
Escola de Engenharia

Departamento de Informática

Miguel Ângelo Abreu de Carvalho

Exploring Mesh Shaders

March 2022



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Miguel Ângelo Abreu de Carvalho

Exploring Mesh Shaders

Master dissertation

Integrated Master's in Informatics Engineering

Dissertation supervised by

António Ramires Fernandes

March 2022

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

LICENSE GRANTED TO USERS OF THIS WORK:



CC BY

<https://creativecommons.org/licenses/by/4.0/>

ACKNOWLEDGEMENTS

To my supervisor, for the availability, support and overall guidance during the course of this thesis.

To my parents, for always giving their all to ensure I had the means to become whoever I desired to be. I'll always strive to make you proud.

To my sister, my friends and my girlfriend, for the needed company and laughter whenever I needed to unwind.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

Every artist is somewhat limited by the mean by which they expose their art. This is also true for the field of [Computer Graphics](#), where there are many limiting factors that developers must go out of their way to avoid. The most limiting of these factors is the computing performance, which directly limits the complexity of what an artist can fabricate in a piece of hardware.

As such, [Computer Graphics](#)' investigators keep an eye out for the improvements made in the hardware department that enables them to introduce more complexity to the scenes they create on their computers.

Three years ago, a novel approach to compute the geometric complexity of [three-dimensional \(3D\)](#) scenes was introduced: Mesh shaders. Mesh shaders pose as an alternative to the traditional geometric processing method and can be a more performant approach to handle specific geometric workloads. Notwithstanding, little attention has been given to these shaders.

Thus, this thesis presents an investigative effort to evaluate the value proposition of these shaders across different scenarios. To do so, this thesis puts Mesh shaders against traditional implementations and measures their differences both in method and performance.

By the end of this thesis, the reader should have a concise understanding of Mesh shaders, but not a clear-cut answer regarding their use. These shaders can provide performance benefits in specific scenarios over the traditional approach, but not without considerable care by the developer. In fact, the flexibility provided by the Mesh shaders' approach gives the developer a significant responsibility regarding their final performance. When incorrectly set up, these shaders can result in mediocre performances compared to those of the traditional pipeline.

Ultimately, these shaders should be used by experienced users intending to avoid specific bottlenecks of the traditional approach. For others, the traditional pipeline offers a more streamlined approach, thoroughly optimised by default.

KEYWORDS Computer Graphics, Mesh Shaders, Performance.

RESUMO

Todos os artistas são de alguma forma limitados pelo meio de exposição da sua arte. Isto não deixa de ser verdade com Computação Gráfica, onde existem vários fatores limitadores que os programadores têm de contornar. Entre estes, o mais impeditivo é a velocidade de computação, que limita diretamente a complexidade da arte que pode ser produzida por uma peça de *hardware*.

Deste modo, os investigadores da área de Computação Gráfica mantêm-se atentos às inovações que ocorrem no campo do *hardware* e lhes permitem introduzir mais complexidade nos cenários que criam.

Há três anos, um método inédito para tratar a complexidade geométrica de cenas tridimensionais foi introduzido: *Mesh shaders*. Os *Mesh shaders* apresentam-se como uma alternativa ao método tradicional de processamento de geometria, que pode obter melhor desempenho em certos cenários geométricos. No entanto, não tem sido dada muita atenção a esta alternativa.

Assim, esta tese apresenta uma investigação destes *shaders* com o intuito de avaliar a sua proposta de valor em diferentes situações. Para o fazer, esta tese irá colocar estes *shaders* frente a frente com os *shaders* tradicionais e medirá as diferenças entre ambos, tanto em desempenho como em método.

No final, o leitor deverá possuir uma ideia coesa sobre os *Mesh shaders*, mas não terá uma perceção binária quanto ao uso dos mesmos. Isto porque estes *shaders* podem oferecer um benefício em termos de desempenho em certas situações, mas requerem cuidados adicionais por parte do programador. Da flexibilidade oferecida pelos *Mesh shaders* advém uma responsabilidade significativa para o programador no que toca ao desempenho final dos mesmos. Quando programados incorretamente, estes *shaders* resultarão num desempenho medíocre comparado ao desempenho oferecido pelo método tradicional.

Fundamentalmente, estes *shaders* deverão ser utilizados por utilizadores mais experientes que pretendem evitar *bottlenecks* específicos do método tradicional. Para todos os outros, o *pipeline* tradicional oferece um método mais simples que possui por predefinição otimizações acentuadas.

PALAVRAS-CHAVE Computação Gráfica, Mesh Shaders, Desempenho.

CONTENTS

I INTRODUCTORY MATERIAL

1	INTRODUCTION	1
1.1	Context	1
1.2	Motivation	1
1.3	Objectives	2
1.4	Structure of the dissertation	2
2	STATE OF THE ART	3
2.1	History of the Rendering Pipeline	3
2.2	Rendering Pipeline Overview	8
2.2.1	Vertex Shader Stage	9
2.2.2	Primitive Assembly Stage	9
2.2.3	Tessellation Shader Stage	9
2.2.4	Geometry Shader Stage	12
2.2.5	Rasterization Stage	12
2.2.6	Fragment Shader Stage	13
2.2.7	Compute Shader	14
2.3	Analysis of Mesh & Task Shaders	15
2.3.1	Alternative Pipeline	15
2.3.2	Purpose of Mesh Shaders	16
2.3.3	OpenGL & GLSL Specification of Mesh Shaders	18
2.3.4	Practical Example with Mesh Shaders	21
2.4	Summary	22

II CORE OF THE DISSERTATION

3	DEVELOPMENT WITH MESH SHADERS	24
3.1	Early Challenges	24
3.1.1	Dimensionality of the Task and Mesh Shaders	24
3.1.2	Pre-processing of Data	25
3.2	Rendering 3D models	26
3.2.1	Lua File Generator	27
3.2.2	Updated Task and Mesh Shaders	30
3.3	Tessellation	30

3.3.1	Non-Fractional Inner Tessellation	31
3.3.2	Fractional & Outer Tessellation	37
3.4	Summary	43
4	CASE STUDIES / EXPERIMENTS	44
4.1	Measuring Performance	44
4.2	Standard Error	45
4.3	Vertex Locality Optimizers	46
4.4	Impact of Wasting Mesh Work Groups	49
4.5	Impact of the local_size parameter	52
4.6	Impact of the max_vertices parameter	55
4.7	Impact of the max_primitives parameter	59
4.8	Rendering 3D models with the Mesh Shader Pipeline	62
4.9	Tessellation with the Mesh Shader Pipeline	65
4.10	Procedural Generation with the Mesh Shader Pipeline	69
4.11	Summary	71
5	CONCLUSIONS AND FUTURE WORK	72
5.1	Conclusions	72
5.2	Prospect for future work	73

LIST OF FIGURES

Figure 1	Rendering Pipeline of the 3dfx Voodoo Graphics Processing Unit (GPU).	4
Figure 2	Rendering Pipeline of the GeForce 256 GPU.	5
Figure 3	Rendering Pipeline supported since the GeForce 3 GPU.	5
Figure 4	Rendering Pipeline supported since the Geforce 6800 GPU.	6
Figure 5	Non-Unified vs Unified shader Model (adapted from Vuong and Pham (2016)).	6
Figure 6	Rendering Pipeline supported since the Geforce 8800 GTX GPU.	7
Figure 7	Rendering Pipeline supported since the Radeon HD 5780 GPU.	7
Figure 8	Rendering Pipeline with Mesh and Task shaders.	8
Figure 9	Example of Sphere Tessellation (Fiedler, 2013).	10
Figure 10	Tessellation shader.	10
Figure 11	Comparison between stitching together different levels of tessellation with and without matching outer levels.	11
Figure 12	Regular Rendering (left) and Wireframe Rendering (right).	12
Figure 13	Rasterization Process.	13
Figure 14	Using Multiple Render Targets (MRTs) for Deferred Shading (Haneda, 2013).	14
Figure 15	Compute shader Pipeline.	15
Figure 16	Rendering Pipeline with Mesh shaders.	16
Figure 17	Representation of a 3D model of a bunny subdivided into differently colored meshlets (Kubisch, 2018).	17
Figure 18	Comparison of the renders when changing the way Mesh shaders are distributed.	22
Figure 19	Proposed meshlet data structure (Kubisch, 2018).	26
Figure 20	Some forms of Tessellation available with Tessellation shaders.	31
Figure 21	Non-fractional Inner Tessellation (14x14).	32
Figure 22	Number of Mesh shader work groups launched based on their configuration.	34
Figure 23	Number of Mesh shader work groups launched based on their configuration.	35
Figure 24	Transition between two levels of tessellation..	38
Figure 25	Naive approach to outer primitive construction.	38
Figure 26	Primitives on the outer levels.	38
Figure 27	Increasing outer levels in the Tessellation shaders.	39
Figure 28	When levels aren't powers of two.	40
Figure 29	Example of optimal distribution of the primitives by the mesh work groups.	41
Figure 30	Example of the distribution method used.	42
Figure 31	Distribution of the bases by the work groups.	42
Figure 32	Powerplant No. Meshlets / Miss Ratio to Performance Correlation.	49

Figure 33	Performance impact of launching more Mesh shader work groups than necessary (i.e., 10000).	50
Figure 34	Squared Approach vs Re-sizable (no per-vertex workload added).	51
Figure 35	Squared Approach vs Re-sizable (with per-vertex workload added).	51
Figure 36	Recorded Performance with each Local Size (Stanford Dragon model).	53
Figure 37	Performance impact of using different values for the local.size parameter in the Stanford Dragon, Powerplant and San Miguel models, respectively.	54
Figure 38	Recorded Performance by Max Vertices value (Lucy model).	56
Figure 39	Recorded Performance by Max Vertices value (Stanford Dragon model).	56
Figure 40	Recorded Performance by Max Vertices value (Powerplant model).	57
Figure 41	Recorded Performance by Max Primitives value (Powerplant model).	58
Figure 42	Recorded Performance by Max Vertices value (Rungholt model).	58
Figure 43	Recorded Performance by Max Primitives value (Lucy model).	60
Figure 44	Recorded Performance by Max Primitives value (Rungholt model).	61
Figure 45	Recorded Performance by Max Primitives value (Powerplant model).	62
Figure 46	Measured Performance with both pipelines for the Sponza (left) and Dragon (right) models.	63
Figure 47	Measured Performance with both pipelines for the San Miguel (left) and Lucy (right) models.	64
Figure 48	Measured Performance with both pipelines for the Rungholt (left) and Powerplant (right) models.	64
Figure 49	Measured performance of both pipelines when tessellating a quad with different tessellation levels: 16 (left), 32 (middle) and 64 (right).	65
Figure 50	Measured performance of both pipelines when tessellating a quad with different tessellation levels: 256 (left), 512 (middle) and 1024 (right).	66
Figure 51	Measured performance of both pipelines when tessellating a quad with different tessellation levels: 2048 (left), 4096 (middle) and 8192 (right).	67
Figure 52	Measured performance of both pipelines when tessellating a quad with different tessellation levels: 2048 (left), 4096 (middle) and 8192 (right). Mesh shader limited to 64 tessellation levels.	67
Figure 53	Render of the Ocean Waves project.	68
Figure 54	Recorded Performance of the Ocean Waves project on the both pipelines with several mesh pipeline configurations.	68
Figure 55	Measured performance of both pipelines when tessellating a quad with different tessellation levels: 2048 (left), 4096 (middle) and 8192 (right). Added heavy per-vertex workload.	69
Figure 56	Measured performance of both pipelines when generating different amounts of grass blades: 65000 (left), 1105000 (middle) and 3315000 (right). For each scenario, different amounts of grass blades per shader instance are emitted.	70

- Figure 57 Measured performance of both pipelines when generating different amounts of grass blades: 1105000 (left) and 3315000 (right). For each scenario, a single grass blade per shader instance is emitted. 70

LIST OF LISTINGS

2.1	Mesh & Task shader Draw Commands in OpenGL.	18
2.2	Indirect Structure for Mesh Draw Commands in OpenGL.	18
2.3	Definition of the Work Group sizes in GLSL (for both Mesh and Task shaders).	19
2.4	Built-in inputs and variables of Task and Mesh shader in GLSL.	19
2.5	Built-in outputs of the Task shader in GLSL.	20
2.6	Built-in outputs of the Mesh shader in GLSL.	20
2.7	Example of User-defined Task shader Outputs.	21
2.8	Example of User-defined Mesh shader Inputs.	21

LIST OF TERMS

data pipeline is a set of data processing elements connected in series, where the output of one element is the input of the next one. [1](#)

debugging is the process of finding and resolving defects or problems that prevent the correct operation of computer programs. [12](#)

instancing is the practice of rendering multiple copies (*i.e.*, instances) of the same object (*e.g.*, trees, grass, etc.) in a scene at once. [9](#)

interpolation is a type of estimation for constructing new data points within the range of a discrete set of known data points. [9](#)

large scale integration refers to an integrated circuit with 500 to 2000 transistors. [3](#)

popping is the undesirable visual effect that occurs when the transition between two different levels of detail is abrupt and obvious to the viewer. [37](#)

RGB is a color model in which red, green, and blue light are added together in various ways to reproduce a broad array of colors. [3](#)

sprite is a two-dimensional bitmap image that is integrated into a larger scene, most often in a [two-dimensional \(2D\)](#) video game. [3](#)

LIST OF ABBREVIATIONS

2D two-dimensional. [xii](#), [1](#), [4](#)

3D three-dimensional. [iv](#), [viii](#), [1](#), [3](#), [4](#), [6](#), [7](#), [17](#), [19](#), [24](#), [26](#), [27](#), [29](#), [44](#), [46](#), [48](#), [54](#), [55](#), [56](#), [58](#), [59](#), [62](#), [65](#), [72](#), [73](#), [74](#)

API Application Programming Interface. [8](#), [9](#), [54](#)

CG Computer Graphics. [iv](#), [1](#), [3](#)

CPU Central Processing Unit. [3](#), [4](#), [8](#), [33](#)

CSV Comma-separated Values. [44](#)

CUDA Compute Unified Device Architecture. [7](#)

FPS frames per second. [45](#), [46](#), [47](#), [52](#), [55](#), [59](#), [62](#), [63](#), [65](#), [66](#), [69](#)

GPGPU General-Purpose Graphics Processing Unit. [7](#)

GPU Graphics Processing Unit. [viii](#), [4](#), [5](#), [6](#), [7](#), [8](#), [25](#), [44](#), [72](#)

GUI Graphical User Interface. [4](#)

ID Identification Number. [9](#), [12](#), [15](#), [18](#), [19](#), [21](#)

LSI Large Scale Integration. *Term:* [large scale integration](#), [3](#)

MRT Multiple Render Targets. [viii](#), [14](#)

OS Operating System. [45](#)

PC Personal Computer. [3](#)

SM Streaming Multiprocessor. [52](#)

SSBO Shader Storage Buffer Object. [25](#), [26](#), [73](#)

TIGA Texas Instruments Graphics Architecture. [3](#)

VAO Vertex Array Object. [25](#), [26](#)

VBO Vertex Buffer Object. [25](#)

XML Extensible Markup Language. [24](#)

Part I

INTRODUCTORY MATERIAL

INTRODUCTION

This chapter introduces the context and theme of this thesis (in [Section 1.1](#)). Furthermore, it also presents the motivation behind it (in [Section 1.2](#)) and the objectives it desires to achieve (in [Section 1.3](#)).

1.1 CONTEXT

In [Computer Graphics \(CG\)](#), the rendering pipeline is the conceptual model that characterizes the process of converting a [three-dimensional \(3D\)](#) scene into a [two-dimensional \(2D\)](#) image. Once the properties of the [3D](#) scene are gathered, they're submitted to this process, which will output a [2D](#) image that can be displayed on a screen. This process is named rendering and it's composed of many sub-processes arranged as a [data pipeline](#), hence the name: rendering pipeline.

Despite being a conceptual model, the rendering pipeline is tightly connected to its hardware implementation. As a matter of fact, the model is structured as a pipeline because that's how specialized hardware initially implemented it.

Due to this intrinsic connection, modifications to the rendering pipeline usually come from the adoption of features introduced by hardware manufacturers.

As such, in 2018, NVIDIA – a distinguished graphics hardware vendor – introduced a promising new feature: Mesh shaders. Mesh shaders have the potential to overhaul the traditional rendering pipeline and are the theme of this thesis.

1.2 MOTIVATION

Not every new feature introduced by hardware vendors ends up being used by developers. There are many factors that come into consideration before those features are adopted, and evaluating those factors is the responsibility of the developer.

However, despite having passed more than three years since Mesh shaders were introduced, there are few independent reviews that evaluate the pros and cons of using Mesh shaders in the rendering process.

Thus, this thesis emerges to evaluate these shaders' purpose in the rendering process by producing an in-depth review of their benefits and drawbacks, when compared with the traditional approach. Hopefully, by the end of this thesis, the reader will be familiarised with Mesh shaders and the value they offer.

1.3 OBJECTIVES

The main goal of this thesis is to offer the reader a coherent and concise analysis of Mesh shaders and the changes they introduce.

To do this, this thesis will put Mesh shaders against traditional implementations in order to evaluate their value proposition. By doing so, this thesis will measure how valuable Mesh shaders are in distinct scenarios, as well as explain how they work along the way.

Hence, this thesis is intended to help developers gain insight into the potential of Mesh shaders, by providing thoroughly tested application scenarios where Mesh shaders are put up against the other available alternatives.

1.4 STRUCTURE OF THE DISSERTATION

This document is divided in five chapters:

1. Introduction ([Chapter 1](#))
2. State of the Art ([Chapter 2](#))
3. Development with Mesh shaders ([Chapter 3](#))
4. Case Studies & Experiments ([Chapter 4](#))
5. Conclusions ([Chapter 5](#))

[Chapter 1](#) introduces the context and theme of this thesis. Furthermore, it also presents the motivation behind it and the objectives it desires to achieve.

[Chapter 2](#)) presents all the contextualization needed to understand the appearance of Mesh and Task shaders.

[Chapter 3](#)) explains all the development which had to be done to be able to investigate the value of Mesh and Task shaders.

[Chapter 4](#)) presents the tests and experiments done to evaluate the value of Mesh and Task shaders across several scenarios.

[Chapter 5](#)) makes known to the reader the conclusions made by the process of investigating Mesh and Task shaders.

STATE OF THE ART

This chapter provides a contextualization of Task and Mesh shaders. Initially, the history of the rendering pipeline will be exposed ([Section 2.1](#)) to establish the context that prompted the appearance of the Mesh and Task shaders. This is followed by an overview of the role of each stage in the pipeline ([Section 2.2](#)). Afterwards, an in-depth examination is done regarding the Mesh and Task shaders ([Section 2.3](#)).

2.1 HISTORY OF THE RENDERING PIPELINE

Over the course of computing history, several hardware accelerators have emerged to perform specialized tasks efficiently outside of the general-purpose [Central Processing Unit \(CPU\)](#). One of the tasks that a [CPU](#) struggled to execute efficiently was displaying complex graphics in a screen, such as those required by video games.

Thus, in the 1970s, many hardware companies started creating specialized hardware to handle these tasks and push the [Computer Graphics \(CG\)](#) industry forward.

In the amusement arcade business, this push resulted in several popular video-games being released (*e.g.* Atari's Pong, Atari's Breakout and Taito's Space Invaders), with each leveraging specialized graphics hardware built into the arcade machine. By 1979, the Namco Galaxian arcade system already featured [sprites](#) and [RGB](#) color.

In the home entertainment industry, Nintendo's Color TV-Game series and the Atari 2600 were sought out consoles that proved there was a market for consumer hardware capable of displaying complex interactive graphics.

The early 1980s brought the NEC μ PD7220 – the first implementation of a [Large Scale Integration \(LSI\)](#) integrated circuit chip for [Personal Computer \(PC\)](#) graphics. This chip allowed manufacturers (*e.g.*, Number Nine Visual Technology) the production of high-performance video graphics cards at a low-cost. These graphic cards could then be added to home computers (*e.g.*, Apple II) to enable them to execute graphical tasks with higher efficiency.

In 1986, Texas Instruments released the TMS34010 – the first fully programmable graphics processor. It could run general-purpose code, but it had a graphics-oriented instruction set. This chip then became the basis of the [Texas Instruments Graphics Architecture \(TIGA\)](#) accelerator cards ([Wikipedia, 2021a](#)).

By the late 1980s, Namco introduced the Namco System 21, which featured an arcade board specifically designed for the processing of [3D](#) polygonal graphics. This introduced the concept of real-time hardware-accelerated [3D](#) and it became the new direction for the industry.

By 1995, 2D Graphical User Interface (GUI) hardware acceleration was supported by most graphics chip makers, with lower-cost fixed-function accelerators surpassing general-purpose graphics coprocessors in performance. This led to the disappearance of these coprocessors from the market.

The first successful consumer-level 3D graphics card was introduced by 3dfx Interactive in 1996 - the 3dfx Voodoo (Akenine-Mller et al., 2018). The Voodoo graphics card was the first fully 3D card and had to be supplemented by a 2D graphics card for day-to-day GUI operations. This approach differed from the industry trend of adding 3D extensions to previous 2D-only graphics cards (e.g., S3 Virge, ATI Rage and Matrox Mystique). Thanks to its focus on 3d rendering, the Voodoo graphics card was able to render the video-game Quake with great quality and performance, which led to its quick adoption (Akenine-Mller et al., 2018). The 3dfx Voodoo implemented a fully fixed-function pipeline, which didn't feature vertex transformations, relegating those tasks to the CPU. A representation of this pipeline is shown in Figure 1.

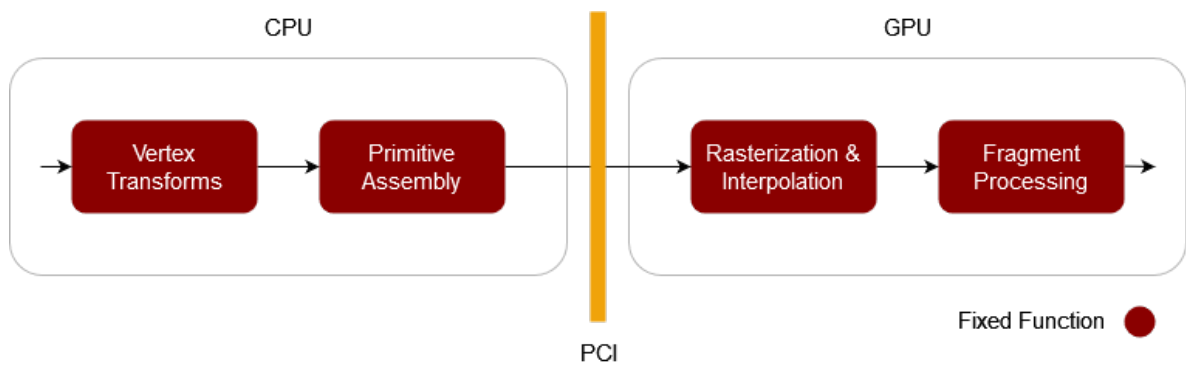


Figure 1: Rendering Pipeline of the 3dfx Voodoo GPU.

In 1998, NVIDIA brought to market the GeForce 256, which was the first graphics card with hardware-accelerated vertex transforms and lighting. This encouraged NVIDIA to call it the first **Graphics Processing Unit (GPU)**, meaning "a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second" (NVIDIA). Moreover, the GeForce 256 provided up to a 50% improvement in frame rate in some games whilst also implementing 2D GUI acceleration. Hence, the GeForce 256 paved the way for a new generation of graphics cards with a more complex rendering pipeline fully implemented in specialized hardware (shown in Figure 2).

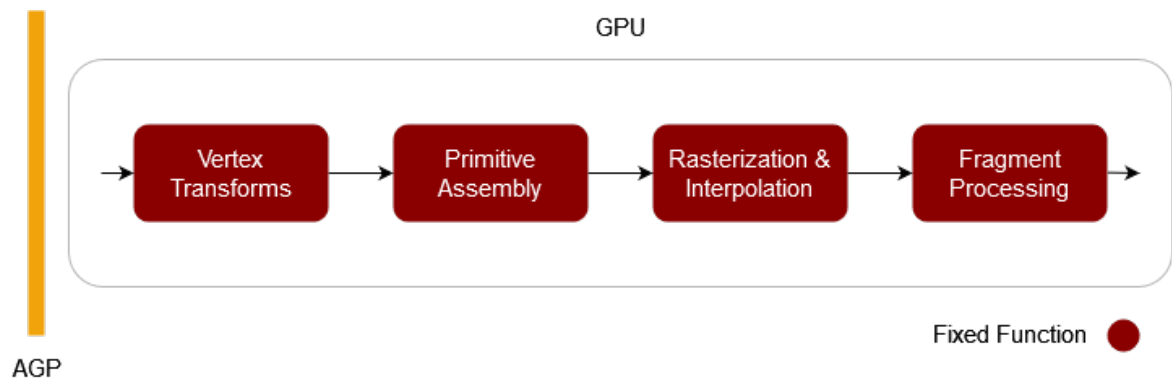


Figure 2: Rendering Pipeline of the GeForce 256 GPU.

The next major step for the rendering pipeline was also introduced by NVIDIA with the release of the GeForce 3 in 2001, which supported programmable Vertex shaders. These shaders were programmed in an assembly-like language that was converted by the drivers into microcode on the fly (Akenine-Miller et al., 2018). However, these shaders missed some features (e.g., flow control), which meant that programmers still couldn't convert most of the software-based shaders they were using in their rendering software (e.g., Pixar's Renderman).

Nonetheless, the addition of these limited programmable shaders paved the way for a future with a more programmable rendering pipeline. Thus, a standard naming scheme was created to uniquely identify the capabilities of these shaders: shader Model 1.0.

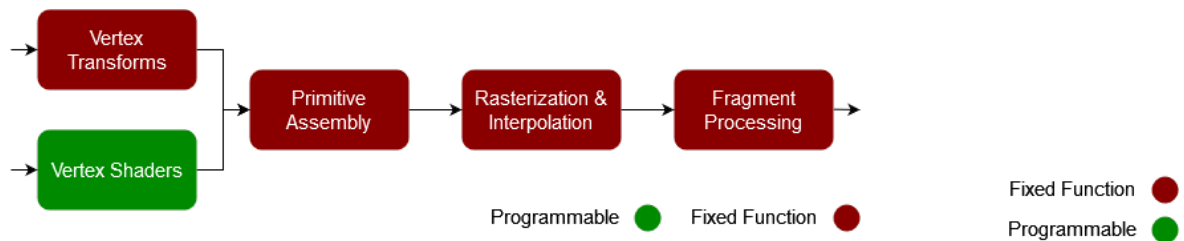


Figure 3: Rendering Pipeline supported since the GeForce 3 GPU.

In 2002, ATI released the Radeon 9700 graphics cards that featured fully-programmable Vertex and Fragment shaders (i.e., shader Model 2.0). These shaders supported non-dynamic flow control (i.e., loops and conditionals) and the Fragment shader supported access texture memory. Two years later, in 2004, NVIDIA brought to market the GeForce 6800 graphics card which featured dynamic flow control and let Vertex shaders access texture memory as well, as defined by the specification of the shader Model 3.0. Hence, by the end of 2004, the graphics pipeline was structured as pictured in Figure 4.

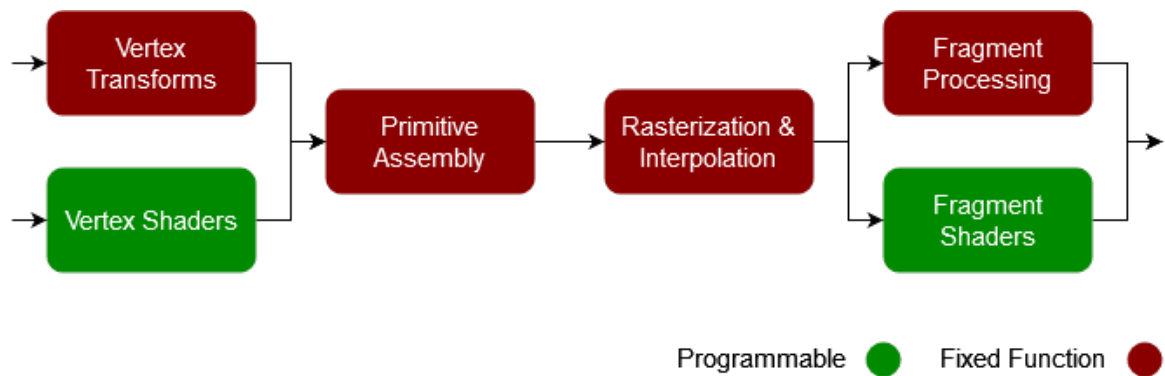


Figure 4: Rendering Pipeline supported since the Geforce 6800 GPU.

The year 2006 brought a major shift in graphics cards' architecture with NVIDIA debuting the Geforce 8800 GTX GPU, which implemented a unified shader architecture. The implementation of an unified shader architecture meant the removal of specialized functional units (*i.e.*, the hardware units designed for processing only specific pipeline stages) and the introduction of a homogeneous collection of universal floating point processors, which could perform a more universal set of tasks (Wikipedia, 2021b). As a consequence, every shader or step in the rendering process would be processed by the same processing units, as shown in Figure 5.

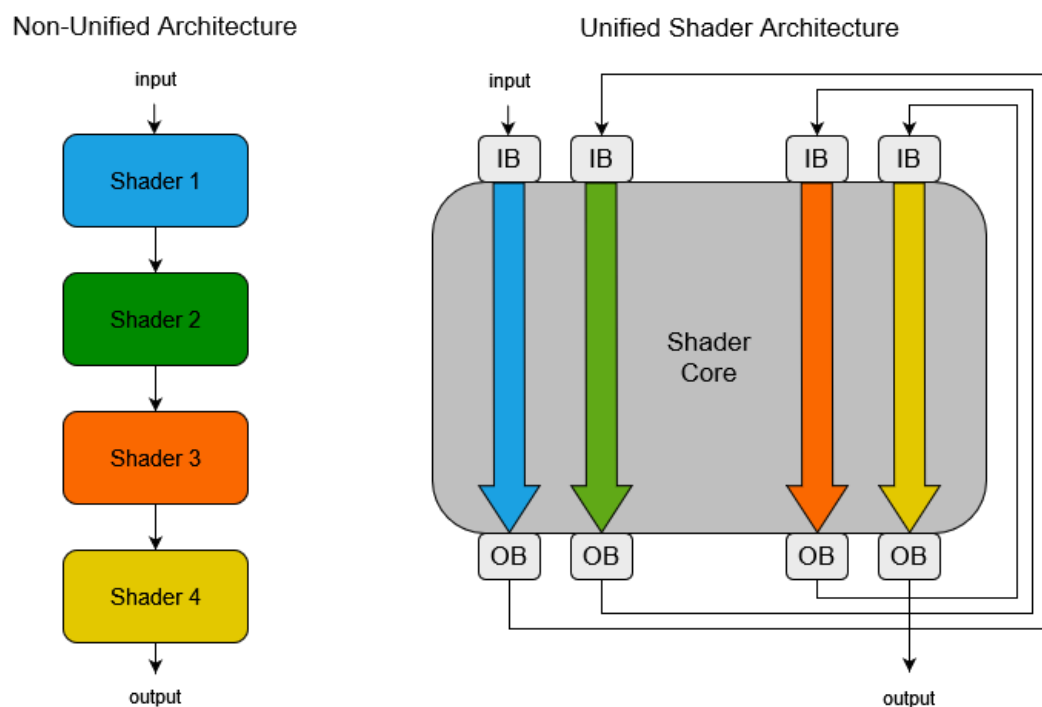


Figure 5: Non-Unified vs Unified shader Model (adapted from Vuong and Pham (2016)).

Fundamentally, the rendering pipeline stopped portraying specialized hardware units in the GPU and became merely a software abstraction of the process of rendering a 3D scene. This change meant that adding steps to the pipeline no longer required the addition of specialized hardware to process them.

Accordingly, the GeForce 8800 GTX introduced in the pipeline a new type of programmable shader – the Geometry shader, an optional shader stage located between the Primitive Assembly and Rasterization stages. With the GeForce 8800 GTX, the Vertex and Fragment shaders became mandatory stages in the rendering pipeline. All of these changes to the rendering pipeline represent the shader Model 4.0, whose conceptual pipeline can be pictured in Figure 6.

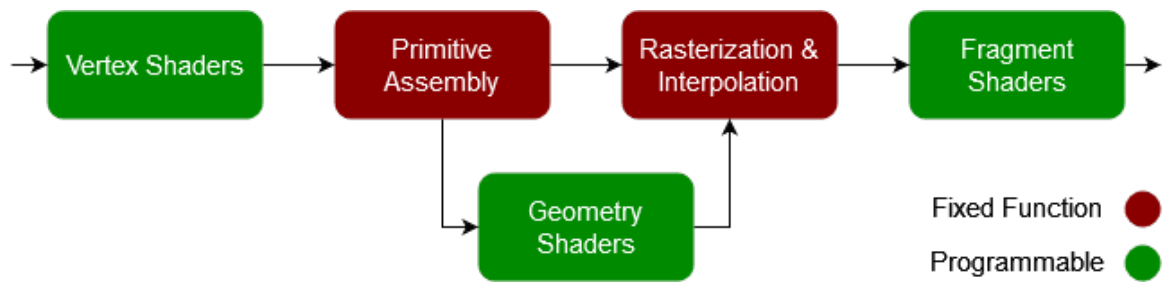


Figure 6: Rendering Pipeline supported since the Geforce 8800 GTX GPU.

To complement this move to more generalized hardware, in 2007 NVIDIA released [Compute Unified Device Architecture \(CUDA\)](#) - a parallel computing platform and programming model for general computing on GPUs (Nvidia, 2017). With CUDA, developers could use the GPU as a [General-Purpose Graphics Processing Unit \(GPGPU\)](#) (i.e., for uses beyond graphics rendering).

The next big rendering pipeline change came in 2009, with the release of ATI's Radeon HD 5870 GPU, which added support for Tessellation shaders. Tessellation shaders were presented as optional shaders composed of two different programmable shaders (i.e., Control and Evaluation) separated by a fixed-function Tessellation Primitive Generator. The updated pipeline with Tessellation shaders is illustrated in Figure 7.

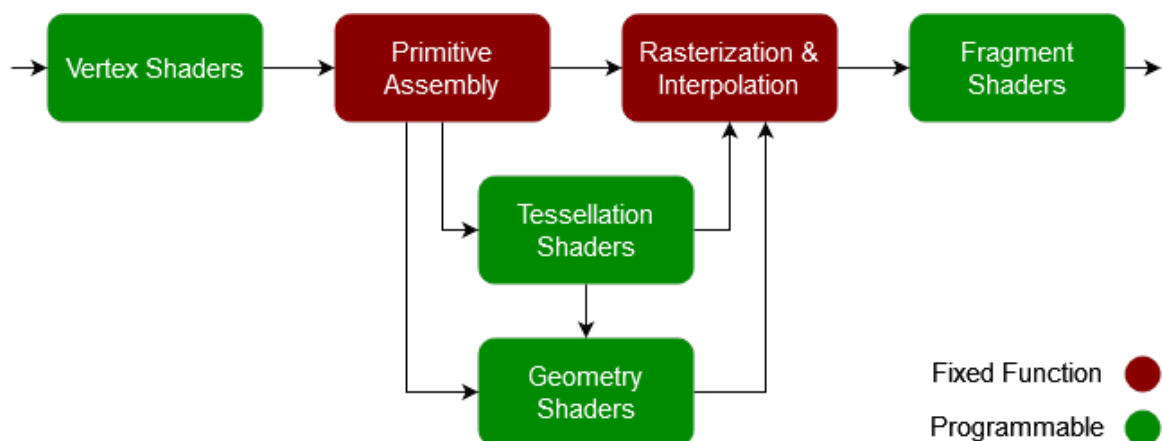


Figure 7: Rendering Pipeline supported since the Radeon HD 5870 GPU.

Besides Tessellation shaders, ATI's Radeon HD 5870 also introduced support to a new type of shaders: Compute shaders. Compute shaders don't exactly fit in the traditional rendering pipeline because they don't implement a clear-cut step in the process of rendering a 3D scene. Compute shaders bring the GPGPU model

to graphics [Application Programming Interfaces \(APIs\)](#), meaning developers can use them in the way they see fit outside of the rendering pipeline.

The addition of the tessellation shaders to the rendering pipeline and the introduction of the general-purpose Compute shaders constitute the changes introduced by shader Model 5.0.

This versatility of compute shaders prompted a stagnant period in the evolution of the rendering pipeline, with developers meeting their needs with custom compute shaders. This stagnant period lasted until 2018 when NVIDIA released the RTX 2080 graphics card, which introduced two new types of programmable shaders: Task and Mesh shaders. Task and Mesh shaders are an alternative to the vertex/primitive stages of the rendering pipeline. The purpose of these shaders is to replace these stages with more compute-like shaders, giving developers more possibilities and ending the reliance on compute shaders for niche algorithms.

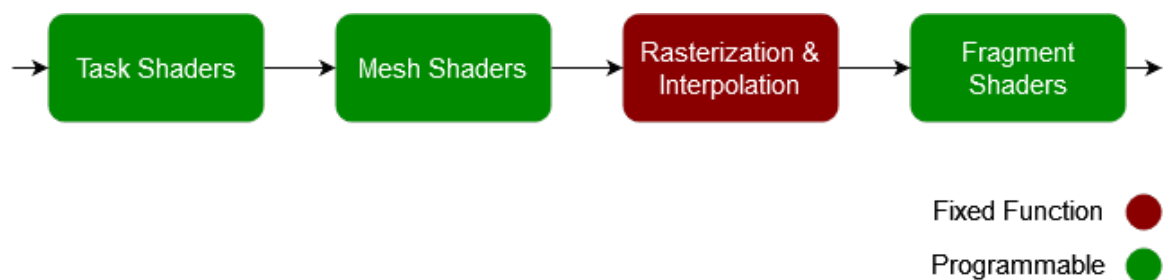


Figure 8: Rendering Pipeline with Mesh and Task shaders.

The arrival of Mesh shaders perpetuates a continuous trend in the evolution of the rendering pipeline. Over time, the rendering pipeline has been evolving away from hard-coded operation and towards increasing flexibility and control. This trend started with the introduction of programmable shader stages and has since been continued with the use of general-purpose shaders stages. More and more, developers want full control over the rendering pipeline and to be limited only by their ingenuity. One can only predict that this trend will carry on in the near future, with more stages of the rendering pipeline being relegated to developer implementations.

2.2 RENDERING PIPELINE OVERVIEW

At the time of writing, the standard rendering pipeline used in computer graphics is more or less as shown in [Figure 7](#). In [Section 2.1](#) the evolution that led to the current rendering pipeline was covered but it gave no insight on the function of each stage. Hence, this section will cover each stage of the rendering pipeline in more depth and explain which parts the developer can tweak.

However, before the first stage of the traditional rendering pipeline can execute, the [GPU](#) has to collect the vertex data sent by the [CPU](#) and feed it to this stage. The structure for this data can be chosen by the developer in the rendering software. This structure is essentially a set of attributes which the developer wants each vertex to have (e.g., position, normal, texture coordinates, etc.). Then, the developer executes a draw command which sends these vertex data structures to the rendering pipeline for primitive building. Primitives are the shapes (e.g., points, lines or triangles) drawn by combining the vertices and the developer must specify beforehand which type of primitive is to be built.

When this data arrives at the rendering pipeline, it is fed to the Vertex shader stage (explored in [Subsection 2.2.1](#)).

2.2.1 Vertex Shader Stage

The Vertex shader is the programmable stage where the developer can perform per-vertex operations. Each Vertex shader instance receives as input the set of attributes of a single vertex provided by the Vertex Specification stage. Each Vertex shader also receives as built-in inputs three [Identification Numbers \(IDs\)](#): one to identify the draw command, one to identify the vertex being processed and one to identify the current instance (see [instancing](#)). Besides inputs, Vertex shaders can also fetch textures and buffers from memory, if needed.

With this information, the developer describes in the shader the operations to be applied to the input vertex. Vertex shaders can't create or destroy vertices, and thus the developer must output attributes concerning a single vertex.

The output data of a Vertex shader is consumed by the next stage of the pipeline: the Primitive Assembly.

2.2.2 Primitive Assembly Stage

The Primitive Assembly Stage is the stage where the stream of vertices produced by the Vertex shader is converted into a sequence of individual base primitives, either to be passed on to a Tessellation shader, a Geometry shader or to the Rasterization stage.

The type of base primitives constructed is dependent on the next stage of the pipeline. If Tessellation shaders are present, the type of primitive built is dependent on the type of patch the Tessellation shaders expect. If there are no Tessellation shaders but there is a Geometry shader, then the type of primitive built is defined by the type of primitive defined as input in the Geometry shader. When none of these shaders is present, the primitive to be constructed for rasterization is defined by the type of `glDraw` command used in the OpenGL [API](#).

2.2.3 Tessellation Shader Stage

The purpose of the Tessellation shader, as the name suggests, is to perform tessellation on the primitives generated from the vertices provided by the Vertex shader. For the uninitiated, tessellation is the process of subdividing primitives into smaller primitives. As such, contrary to what happens with Vertex shaders, the Tessellation shader receives as input a finite set of primitives, called a patch. To generate primitives from said patch, new vertices are generated via the [interpolation](#) of the patch vertices.

Usually, the purpose of the Tessellation shader stage is to add detail to a model (as seen in [Figure 9](#)). Different levels of detail can be set to each primitive depending on the situation (*e.g.*, adding detail to objects far away from the camera produces diminishing returns).

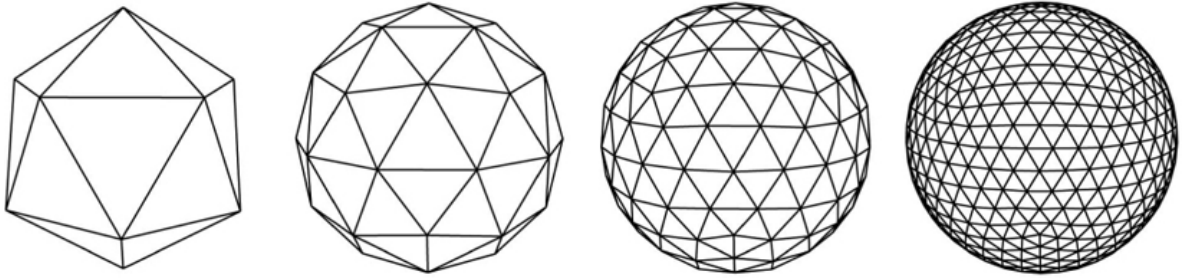


Figure 9: Example of Sphere Tessellation (Fiedler, 2013).

The Tessellation shader stage is composed of three substages: Tessellation Control shader (Subsubsection 2.2.3), Primitive Generation (Subsubsection 2.2.3) and Tessellation Evaluation shader (Subsubsection 2.2.3). These stages are arranged as illustrated in Figure 10.

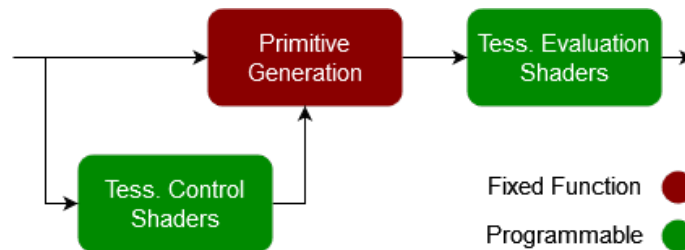


Figure 10: Tessellation shader.

Tessellation Control Shader Stage

The Tessellation Control shader is the optional programmable shader within the broader Tessellation stage. If present, the Tessellation Control shader is the first stage to be executed. If absent, predefined values (*i.e.*, tessellation levels) are passed on to the Primitive Generation Stage (Subsubsection 2.2.3).

The purpose of the Tessellation Control shader is to establish the amount of tessellation to apply to the input patch. To do this, the developer must set the levels of tessellation. The higher the levels of tessellation chosen, the higher will be the number of subdivisions created within said patch.

These levels are separated into inner levels and outer levels. The outer levels allow the developer to choose different levels of tessellation for each edge of the patch, independently of the inner level of tessellation. This allows the developer to connect two patches with different inner levels of tessellation, as long as the edge by which they connect shares the same outer level in each patch (shown in Figure 11b). If the developer joins two patches with non-matching outer levels, visual artifacts may appear in certain scenarios (shown in Figure 11a).

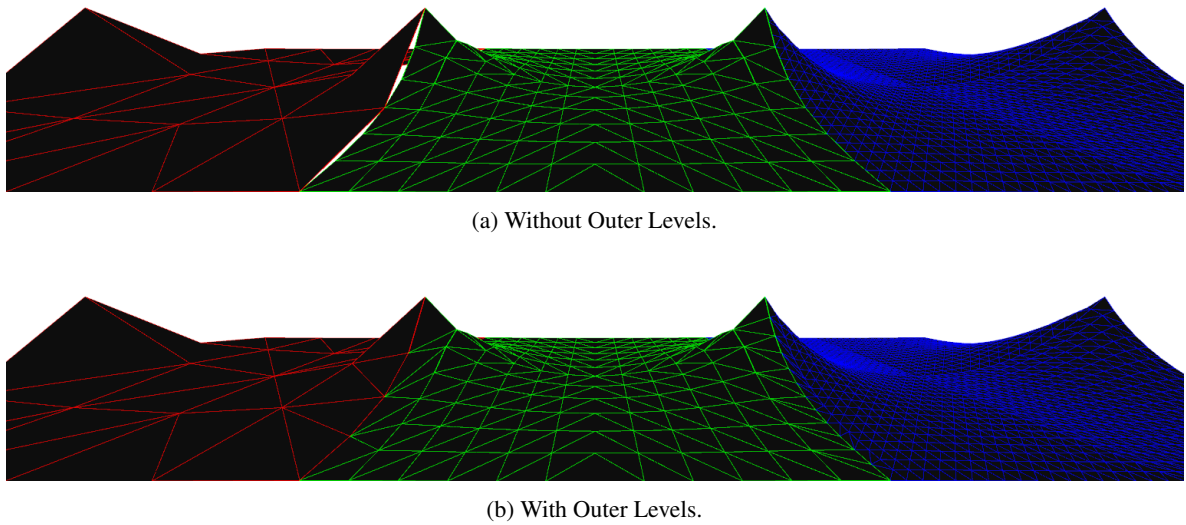


Figure 11: Comparison between stitching together different levels of tessellation with and without matching outer levels.

The developer may also choose to set the levels of tessellation of the primitive to zero so that the Primitive Generation stage discards the primitive. Discarding a primitive in this stage of the pipeline is useful as further stages no longer need to expend resources to process it. For instance, if a primitive is out of view, there's no point tessellating it or doing further work with it. As such, this stage can be used to check if the primitive is out of view (*i.e.*, if it's outside the camera's view frustum) and discard it if so. Avoiding rendering primitives that won't appear on screen is called culling and is a technique used to spare the hardware from doing unnecessary work, which results in a smaller rendering time.

Primitive Generation Stage

The Primitive Generation stage is the fixed-function step of the Tessellation shader stage that's in charge of generating tessellation coordinates for the primitive. These tessellation coordinates will be used by the Tessellation Evaluation shader to interpolate the vertices of the new primitives.

The tessellation coordinates generated by this stage are dependent on the type of primitive being tessellated (*e.g.*, interpolating in a line requires a single coordinate, whilst in a quadrilateral two are needed). The Primitive Generation stage does not perform any interpolation and relegates that job to the Tessellation Evaluation shader.

Tessellation Evaluation Shader

The Tessellation Evaluation shader comes last in the broader Tessellation shader Stage. Since the Tessellation Control shader stage is optional, it's the presence of the Tessellation Evaluation shader that determines if the Tessellation shader Stage is executed in the pipeline.

As mentioned above, the Tessellation Evaluation shader is tasked with generating the attributes of the new vertices, by using the tessellation coordinates provided by the Primitive Generation stage.

As such, the Tessellation Evaluation shader executes once for each tessellation coordinate. Besides tessellation coordinates, the Tessellation Evaluation shader may also provide attributes for each vertex.

2.2.4 Geometry Shader Stage

The Geometry shader is the optional programmable shader which resides between the Primitive Assembly (or the Tessellation shader, if present) and the Rasterization Stage.

The purpose of the Geometry shader is to allow the developer to manage the primitives generated by the previous stages of the pipeline. As such, each Geometry shader is in charge of a single primitive and has as inputs the attributes of each one of the vertices that form said primitive.

The Geometry shader differs from the remainder of the pipeline stages as it is the only one capable of generating primitives arbitrarily. This is due to the Geometry shader allowing for instancing and the choice of a different primitive type.

Thanks to instancing, the developer can execute the same Geometry shader multiple times for the same primitive and use an **ID** to produce different results each time. Additionally, the developer can choose an arbitrary type of primitive to output. These two features allow the developer to both heavily modify the input geometry and generate completely new geometry on the fly.

A popular use of the Geometry shader is to view the scene in Wireframe mode (exemplified in Figure 10) to check the geometry of the scene. By using a Geometry shader, this task can be done by simply outputting a line primitive for each edge of the input triangle primitive.

Notwithstanding its usefulness, the Geometry shader incurs in a heavy performance cost that few of its uses can compensate for. Thus, the Geometry shader ends up being mostly used in situations where the performance is not vital (e.g., [debugging](#)) or when no other can solve the desired task as elegantly (e.g., voxelization).

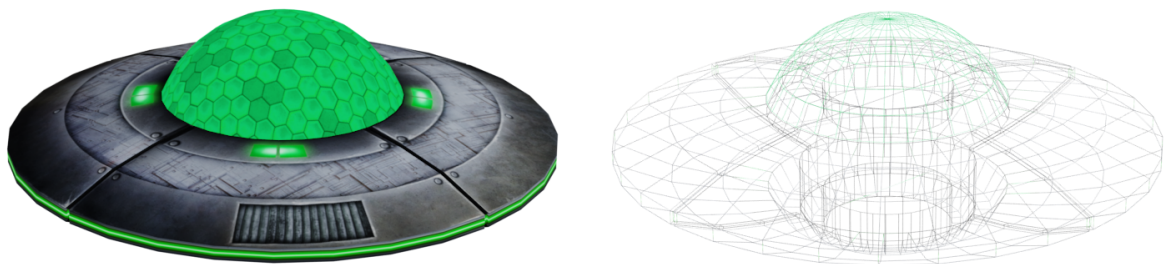


Figure 12: Regular Rendering (left) and Wireframe Rendering (right).

2.2.5 Rasterization Stage

The Rasterization stage is the fixed-function stage whose purpose is to rasterize the primitives that reached this stage. The rasterization process has a different set of rules for each one of the base primitives (*i.e.*, points, lines and triangles) but the result of rasterizing a primitive is always a sequence of fragments.

A fragment is the set of the data that corresponds to a single pixel in the output Frame Buffer. This set includes the position of the fragment on the screen, the depth of the fragment and the arbitrary data that previous shaders outputted. As the output of previous shaders describes the vertices of the primitive being rasterized, the attributes of these vertices have to be interpolated for each fragment, depending on the location of said fragment. The type of interpolation that occurs in this stage can be chosen by the developer in the Fragment shader.

To help visualize the process of rasterization, a rudimentary example of triangle rasterization is shown in Figure 13.

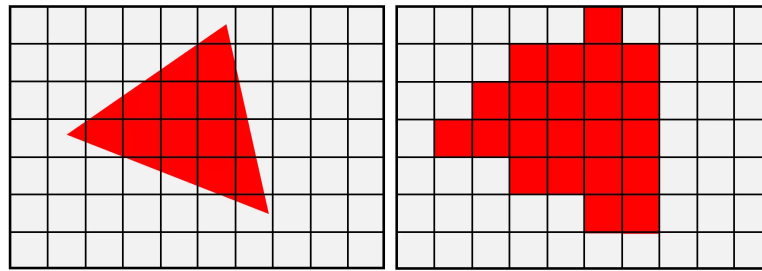


Figure 13: Rasterization Process.

2.2.6 Fragment Shader Stage

The Fragment shader is the programmable stage where the developer performs any per-fragment operations it desires. The Fragment shader is analogous to the Vertex shader, but operates on a per-fragment basis instead of a per-vertex one. As such, each fragment shader receives as input the attributes of a single fragment generated by the Rasterization Stage.

It's up to the developer to then choose how to use this data to compute the output of the fragment shader. This output is usually the color of the fragment to be stored in the color buffer. The color buffer is a part of the collection of buffers called Frame Buffer, which also contains the depth buffer. However, the Fragment shader can also be used to output to other types of buffers besides the color buffer. The developer may choose to output to a render target, whose purpose isn't to be printed in the screen but to be consulted later on. In Figure 14 we can see the popular use for rendering targets: Deferred Shading.

Deferred Shading differs from regular shading because no shading is actually performed in the first pass of the pipeline. Instead, on the first pass, the data that is required for shading computation (e.g., positions, normals, etc.) is gathered into render targets and in the second pass, a fragment shader computes the color for each pixel using the information of said render targets. Figure 14 shows the visualization of the render targets and the result of combining them for the final result.

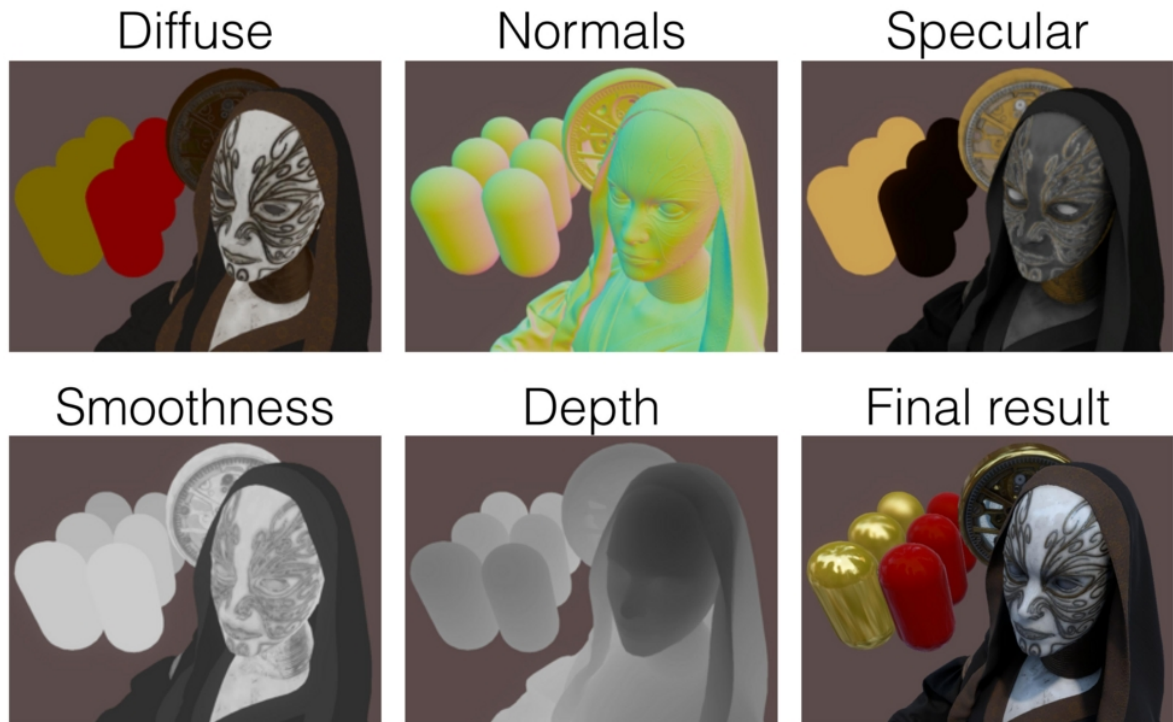


Figure 14: Using [Multiple Render Targets \(MRTs\)](#) for Deferred Shading ([Haneda, 2013](#)).

2.2.7 Compute Shader

The Compute shader is a programmable shader that's not part of the traditional rendering pipeline. Every stage in the rendering pipeline has a well-defined place and purpose. This is not the case with Compute shaders, which were designed for computing arbitrary information. As such, Compute shaders belong to a different pipeline as shown in [Figure 15](#).

Besides belonging to a different type of pipeline, Compute shaders also differ from the other shaders in the frequency of execution, as they don't have a predefined frequency to run at. Whilst, for example, Vertex shaders execute once per vertex, Compute shaders' frequency is up to the developer (*i.e.*, the number of Compute shader executions is defined by the function used to execute the compute operation ([OpenGL Wiki, 2019](#))).

On top of that, Compute shaders have no user-defined inputs or predefined outputs, having to store their outputs in textures or buffers. Yet, Compute shaders have built-in inputs which define the "space" in which they're operating in.

When a developer invokes the execution of a Compute shader, the developer dispatches a certain number of groups of threads, with each thread executing once the contents of the Compute shader (as shown in [Figure 15](#)). Additionally, the developer can also define the size of the thread groups within the Compute shader.

The grouping of threads is to enable the threads in a group to share certain variables, meaning all threads in a group can see and modify said variables. The grouping of threads also allows the developer to use commands to synchronize the parallel execution of threads in the same group.

Each individual thread receives as built-in inputs the number of thread groups dispatched and three [Identification Numbers](#) (IDs). One ID identifies the group in which the thread is being executed in, another differentiates the thread within the group and the last identifies the thread from all other threads, including the ones from other thread groups. The thread also has access to the size of the group as it is a defined constant within the shader.

Since Compute shaders have no implicit connection to the rendering pipeline, developers must use memory objects (e.g., textures or buffers) to communicate with it. Compute shaders must read from memory the outputs of the rendering pipeline stages and write to memory any information they want to be accessed by the rendering pipeline.

Nevertheless, the sheer programmability of Compute shaders inspired developers to use them in a manifold of ways previously not done in the graphics hardware.

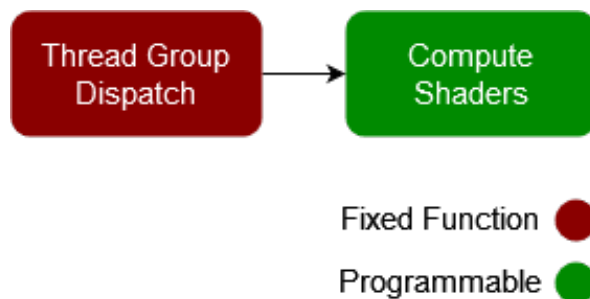


Figure 15: Compute shader Pipeline.

2.3 ANALYSIS OF MESH & TASK SHADERS

In this section, Mesh and Task shaders will be examined. In [Subsection 2.3.1](#), the changes introduced by Mesh and Task shaders are explored. In [Subsection 2.3.2](#), the purpose for Mesh and Task shaders is explained. Finally, in [Subsection 2.3.3](#), the OpenGL & GLSL specification of Mesh and Task shaders is covered.

2.3.1 *Alternative Pipeline*

Mesh and Task shaders are the two new programmable shaders which come as an alternative to every stage of the rendering pipeline until the Primitive Assembly and Rasterization stages. These shaders share many similarities with Compute shaders, as they work in groups of cooperative threads managed by the developer. These threads are then used to compute the geometry of the scene and feed it directly to the next stages. By examining [Figure 16](#), one can see the several changes induced by the introduction of the compute-like Mesh shaders to the traditional rendering pipeline (shown in [Figure 7](#)).

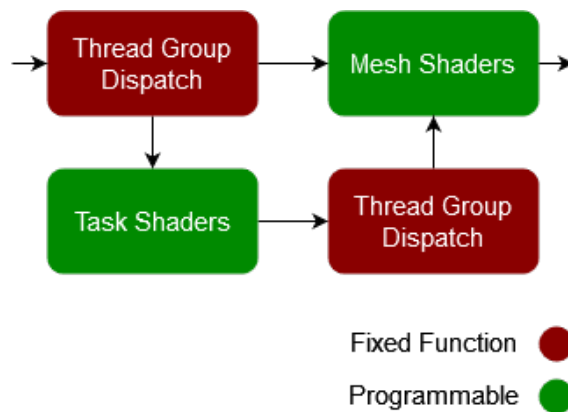


Figure 16: Rendering Pipeline with Mesh shaders.

Firstly, Vertex, Tessellation and Geometry shaders are not present when a pipeline contains Mesh shaders and optionally Task shaders. This means that every previous task done by the former shaders (*e.g.*, tessellation or primitive generation) must be transported to the Mesh and Task shaders.

Secondly, the Mesh shader must retrieve the vertex attributes from memory, just like Compute shaders would. This implication can be both a boon and bane, as is now the job of the developer to use the memory in a responsible manner.

Lastly, contrary to Compute shaders, Mesh shaders feed the transformed vertices directly to the Primitive Assembly stage. Thus, by using Mesh shaders to compute any geometry calculations previously being done in Compute shaders for performance reasons, the developer is removing the delay introduced by memory access and theoretically improving the performance of its rendering process.

2.3.2 Purpose of Mesh Shaders

Mesh and Task shaders come as a direct response to developers who were already using the Compute shader to somehow handle the geometry of a scene. In the traditional pipeline there are some limitations and bottlenecks in the way the geometry is handled, which developers were already trying to solve by using Compute shaders.

One of these bottlenecks is a fixed-function block called the Primitive Distributor. The Primitive Distributor occurs before the Vertex shader and is in charge of distributing the geometry in chunks to be handled by the Vertex shaders. To do this, the Primitive Distributor reads the index buffer serially, makes batches of vertices and distributes them by the Vertex shaders. The batches created by this fixed-function unit are well optimized to take advantage of the hardware. Additionally, this process happens every frame, which could be wasteful for static topologies, and is done in a serial manner, thus not taking advantage of the massively parallel hardware.

To solve this bottleneck, Mesh and Task shaders expect these batches (*i.e.*, *meshlets*) to be pre-computed and not calculated every frame. These *meshlets* are then accessed by the Mesh and Task shaders in parallel via memory buffers. Although this solves the bottleneck, it puts the responsibility of creating the *meshlets* in the hands of the developer, which now has to do the optimizations previously done by the Primitive Distributor. A visualization of the *meshlets* of a model can be seen in [Figure 17](#).



Figure 17: Representation of a 3D model of a bunny subdivided into differently colored *meshlets* (Kubisch, 2018).

Other limitations of the traditional rendering pipeline come by the limitations of each shader and their ordering in the pipeline. As seen in the Section 2.2, each shader works at a certain frequency and has limited inputs and outputs. Additionally, they must be ordered in a certain way in the pipeline. This means that processes usually have to be done in a certain order or be done in multiple passes.

Let's see culling as an example:

1. The Vertex shader fetches its vertex and all of its attributes, computes all the necessary per-vertex operations and then feeds the vertex to the tessellation shader.
2. The Tessellation shader receives the primitive as input and notices the primitive should to be discarded.

In the example above, both the fetch of the attributes and the per-vertex operations – not including the transformations – were wasted computation. To solve this problem, it's possible to use the Compute shader to do some pre-culling before advancing to the traditional rendering pipeline. However, Compute shaders don't fit in the same pipeline and require memory accesses to the feed data across the pipelines.

Mesh and Task shaders are uniquely tailored to fix this problem, as Task shaders can perform pre-culling, dispatch Mesh shader work groups accordingly, with each Mesh shader making the per-vertex computations only for primitives which can't be culled beforehand. This is all done in a single pipeline, thus taking advantage of the hardware's cache to pass the data between the stages. In this case, Mesh and Task shaders allow for less geometry computation and less memory accesses, which could ultimately imply better performance.

Nonetheless, the theoretical improvements in performance of the Mesh pipeline have to be tested in practice. As such, later in the thesis several use cases are investigated in more detail to assess how valuable Mesh and Task shaders truly are (*i.e.*, if using Mesh shaders results in better performance than using traditional shaders).

2.3.3 OpenGL & GLSL Specification of Mesh Shaders

Looking at the execution of Mesh and Task shaders, similarly to Compute shaders, the developer must choose the number of work groups to dispatch. If the developer chooses to use Task shaders in its pipeline, then these threads will run as Task shaders. Otherwise, these threads will execute as Mesh shaders. In OpenGL, the following commands available to launch a certain number of Task or Mesh shaders:

```
void DrawMeshTasksNV(uint first, uint count);

void DrawMeshTasksIndirectNV(intptr indirect);

void MultiDrawMeshTasksIndirectNV(intptr indirect,
                                   sizei drawcount,
                                   sizei stride);

void MultiDrawMeshTasksIndirectCountNV(intptr indirect,
                                         intptr drawcount,
                                         sizei maxdrawcount,
                                         sizei stride);
```

Listing 2.1: Mesh & Task shader Draw Commands in OpenGL.

The `DrawMeshTasksNV` command launches `count` work groups of Task or Mesh shaders, with the work group `ID` of each being inclusively within the range of `first` and `first+count-1`.

The `DrawMeshTasksIndirectNV` command is analogous in behaviour to the `DrawMeshTasksNV` command but uses the `indirect` argument as an offset to the `DRAW_INDIRECT_BUFFER`, which must store in said offset a structure containing both the `first` and `count` values (as shown in the [Listing 2.2](#)).

```
typedef struct {
    uint count;
    uint first;
} DrawMeshTasksIndirectCommandNV;
```

Listing 2.2: Indirect Structure for Mesh Draw Commands in OpenGL.

The `MultiDrawMeshTasksIndirectNV` command is analogous to the `DrawMeshTasksIndirectNV` command, but expects the `DRAW_INDIRECT` buffer to contain in the `indirect` offset `drawcount` structures of data. The distance between each structure is provided in the `stride` argument.

The `MultiDrawMeshTasksIndirectCountNV` command is similar to the `MultiDrawMeshTasksIndirectNV` command, but the `drawcount` argument no longer specifies the number of structures of data stored directly. Instead, the `drawcount` represents an offset to the `PARAMETER_BUFFER_ARB` bound buffer which stores the number of structures of data. The `maxdrawcount` argument is the higher limit that the stored number of structures can achieve.

All of these draw commands are limited by the amount of work groups the hardware can dispatch. The maximum number of task or mesh shader work groups that may be dispatched at one time can be obtained by calling the `GetIntegeriv` command with the argument `MAX_DRAW_MESH_TASKS_COUNT_NV`.

Besides the number of work groups it dispatches, the developer can also choose the size of the work groups. This is done within the Task and Mesh shaders. To do this, the developer must use the layout qualifier `local_size_x`, as shown in the Listing 2.3 where a work group size of thirty-two is chosen. GLSL also supports the layout qualifiers `local_size_y` and `local_size_z`, but those currently are limited to a size of one, as there's no support for multi-dimensional work groups provided for Mesh and Task shaders. To access the size chosen in the layout qualifiers, both Mesh and Task shaders can use the built-in constant variable `gl_WorkGroupSize`, shown in the Listing 2.4.

```
layout(local_size_x=32) in;
```

Listing 2.3: Definition of the Work Group sizes in GLSL (for both Mesh and Task shaders).

Since both Task shaders and Mesh shaders can be executed by the initial thread groups, they share the same built-in inputs and variables. Each Task and Mesh shader knows the size of its thread group and receives as inputs three IDs. Similarly to the Compute shaders IDs, one ID (*i.e.*, the `gl_WorkGroupID`) identifies the group in which the thread is being executed in, another (*i.e.*, the `gl_LocalInvocationID`) differentiates the thread within the group and the last (*i.e.*, the `gl_GlobalInvocationID`) identifies the thread from all other threads, including the ones from other thread groups. These three IDs are 3D values, but as seen above both the number of work groups and the size of the work groups are defined by a single value, which means only the first value in the 3D array can be used to identify the thread. GLSL also provides a fourth built-in ID (*i.e.*, the `gl_LocalInvocationIndex`), which simply serves as a one-dimensional representation of the `gl_LocalInvocationID`. The definition of all these IDs can be seen in the Listing 2.4.

Additionally, Mesh shaders and Task shaders include infrastructure to allow rendering for multiple views (*e.g.*, left and right eye views for stereoscopic rendering). To do this the shaders receive as input the number of views chosen by the developer (*i.e.*, the `gl_MeshViewCountNV` built-in input) and an ID for the view that each thread will be rendering for (*i.e.*, the `gl_MeshViewIndicesNV` built-in array). The developer can then use the IDs provided for each view to emit per-view attributes for the geometry using the `perviewNV` storage qualifier. Both of these view inputs can be seen in the Listing 2.4.

```
const uvec3 gl_WorkGroupSize;
in uvec3 gl_WorkGroupID;
in uvec3 gl_LocalInvocationID;
in uvec3 gl_GlobalInvocationID;
in uint gl_LocalInvocationIndex;
in uint gl_MeshViewCountNV;
in uint gl_MeshViewIndicesNV[];
```

Listing 2.4: Built-in inputs and variables of Task and Mesh shader in GLSL.

The similarities of Mesh shaders and Task shaders end when comparing the outputs of both.

As a Task shader's purpose is to choose the number of work groups to dispatch for Mesh shader code execution, the Task shader has a single built-in output that defines the number of thread groups to be launched (*i.e.*, `gl_TaskCountNV`). Each Task shader decides how many thread groups it launches and may choose to launch none at all. The definition of the built-in output that allows this is shown in the [Listing 2.5](#).

```
out uint gl_TaskCountNV;
```

Listing 2.5: Built-in outputs of the Task shader in GLSL.

As a Mesh shader's purpose is to output the geometry data for the Primitive Assembly stage, the Primitive Assembly stage must be able to generate primitives from the Mesh shaders' outputs. Thus, each Mesh shader must define initially the type of primitives it will generate – in a layout qualifier with points, lines or triangles – and the maximum vertices and primitives that can be generated by the work group – in the `max_vertices` and `max_primitives` layout qualifiers.

In addition to those, each Mesh shader must indicate the number of primitives it will generate – in the `gl_PrimitiveCountNV` –, the vertices that are used to built those primitives – in the built-in array `gl_MeshVerticesNV` – and the indexes of those vertices in the order of construction of those primitives – via the built-in array `gl_PrimitiveIndicesNV`. As each Mesh shader in a thread groups works for the same geometry, these outputs can be filled by only one of the threads or all of them, with the desired behaviour being all threads filling different parts of those outputs and sharing the work equitably. Both the layout qualifier for the primitive types and the other mandatory built-in outputs are shown in the [Listing 2.6](#).

```
layout(triangles) out;
layout(max_vertices=64, max_primitives=126) out;

out uint gl_PrimitiveCountNV;
out uint gl_PrimitiveIndicesNV[];

out gl_MeshPerVertexNV {
    vec4 gl_Position;
} gl_MeshVerticesNV[];
```

Listing 2.6: Built-in outputs of the Mesh shader in GLSL.

Apart from the built-in outputs, Mesh and Task shaders can also set custom outputs that are fed to the next stage of the pipeline.

For the Task shader, the developer must encompass those outputs in a single output structure qualified by `taskNV`. These user-defined outputs of the Task shader will serve as custom outputs for the Mesh shaders it spawns. [Listing 2.7](#) shown an example of how to set user-defined outputs in a Task shader and [Listing 2.8](#) shows how those user-defined outputs are expressed as inputs in the Mesh shader.

```
taskNV out Task {
    uint baseID;
};
```

Listing 2.7: Example of User-defined Task shader Outputs.

```
taskNV in Task {
    uint baseID;
};
```

Listing 2.8: Example of User-defined Mesh shader Inputs.

2.3.4 Practical Example with Mesh Shaders

To first start rendering with Mesh shaders in OpenGL & GLSL, following the specification explained in [Subsection 2.3.3](#) should suffice. However, to further deepen the reader's knowledge of Mesh shaders, this section provides a quick practical example of how each parameter affects the rendering process.

The practical example chosen was the simple task of rendering a cube. There are several approaches to this problem, but with the purpose of better understanding the peculiarities of rendering with Mesh shaders, let's follow the following steps:

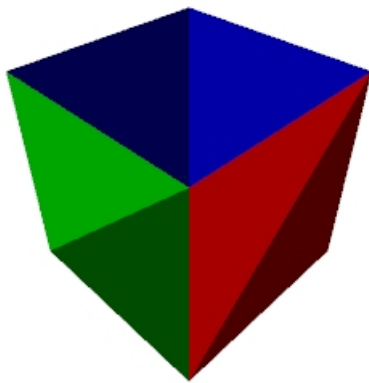
1. Execute the command `voidDrawMeshTasksNV(uint FIRST, uint COUNT)`. For this example, `FIRST` should be set to zero, as in this case there's no advantage in starting the group IDs with another value, and `COUNT` to six, so that each work group takes care of a face of the cube. Effectively, as no Task shader is present in this example, setting the `COUNT` value establishes the number of *meshlets* to be drawn. In this first example, a *meshlet* will correspond to a face of the cube.
2. Set the `layout(local_size_x = GROUP_SIZE) in` with `GROUP_SIZE` being the chosen group size. Following the example above, as each work group handles a face of the cube, the group size can be set of two, so that each thread is tasked with emitting a single triangle.
3. Set the `layout(PRIMITIVE_TYPE, max_vertices = MAX_VERTICES, max_primitives = MAX_PRIMITIVES) out`. `MAX_VERTICES` and `MAX_PRIMITIVES` correspond to the maximum dimensions the *meshlet* emitted by a work group can have, and the `PRIMITIVE_TYPE` indicates the type of primitive to be drawn for said *meshlet*. In this example, each *meshlet* corresponds to a solid face of a cube and, as such, `PRIMITIVE_TYPE` must be set to triangles. As each face of the cube has four vertices and two triangles, `MAX_VERTICES` and `MAX_PRIMITIVES` must be set to these values respectively.
4. Set the `gl_PrimitiveCountNV = PRIMITIVE_COUNT`. The `PRIMITIVE_COUNT` corresponds to the number of primitives the *meshlet* being drawn actual has. In this example, our work is evenly distributed and each work group produces *meshlets* of equal sizes and as such, the number of primitives of each *meshlet* is the same as the maximum previously defined (*i.e.*, two primitives).
5. Emit the *meshlet's* vertex positions. To do this, the vertices of the *meshlet* should be distributed by the threads in the work group, so that each work group cooperatively fills the `gl_MeshVerticesNV` array with the positions of the *meshlet's* vertices. In this case, every thread in the work group will set four vertex positions, as each face only needs four vertices so it can be drawn.

6. Emit the *meshlet*'s primitive indices. Similarly to the vertices, the primitive indices should also be distributed by all the threads launched, so that each thread cooperatively builds a portion of the *meshlet*. In this example, each thread will cover three primitive indices which correspond to a single primitive.
7. To help visualize what each thread contributed to, a per-primitive color (using the `perPrimitiveNV` storage modifier) can also be set so the fragment shader paints each face of the cube with a different color, with each of the triangles of a face also being slightly different in brightness.

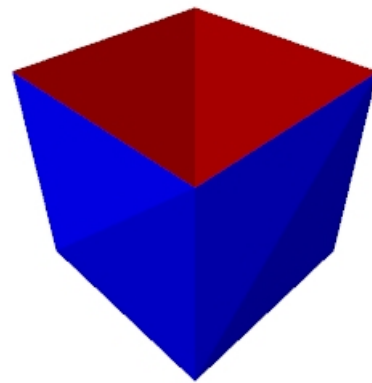
By doing the steps described above, one should produce a Mesh shader which produces the render shown in [Figure 18a](#). As expected, each work group handled a face of the cube and each group thread handled a primitive in said face.

However, as mentioned above, there are several ways of tackling this problem and with Mesh shaders we can also do it differently. [Figure 18b](#) shows the result of altering the code so it dispatches only two thread groups, each with six threads. With this modification, there's now only two colors in the cube but six shades for each color, as expected.

Measuring the performance differences between the two approaches doesn't provide meaningful results, as the amount of geometry is very low. However, with more demanding tasks, these different approaches affect the final performance of the pipeline. As such, the next sections explain the mechanisms needed to generate higher geometry with Mesh shaders, allowing for meaningful performance comparisons of both pipelines.



(a) Six two-threaded work groups.



(b) Two six-threaded work groups.

Figure 18: Comparison of the renders when changing the way Mesh shaders are distributed.

2.4 SUMMARY

It's clear that Mesh and Task shaders present quite a significant change to the status quo of the rendering pipeline. Thus, it's prudent to make an analysis on these shaders to better understand if this significant change in the process is justified.

Part II

CORE OF THE DISSERTATION

DEVELOPMENT WITH MESH SHADERS

This chapter encompasses the development made during this thesis which allowed the attainment of the results presented in [Chapter 4](#). [Section 3.1](#) explores the distinct challenges faced when developing with Mesh and Task shaders instead of the traditional shaders. [Section 3.2](#) expresses the steps followed to achieve the rendering of 3D models with Mesh and Task shaders. [Section 3.3](#) describes the development needed to achieve a tessellation algorithm with Mesh and Task shaders, comparable to the Tessellation shaders' tessellation.

The development platform chosen for this thesis was the Nau 3D engine ([Fernandes, 2014](#)). Nau allows the swift creation of 3D shader pipelines written in GLSL through custom instructions written in an [Extensible Markup Language \(XML\)](#) document. As Nau only supports OpenGL at the backend, Mesh and Task shaders were only evaluated within the OpenGL environment, with other backends such as DirectX being unexplored. Nau also supports Lua scripting to extend its functionality and has built-in profiling tools which can be used for in-depth performance measures.

The source code produced during this development can be found in <https://github.com/maacarrvalho/exploring-mesh-shaders>.

3.1 EARLY CHALLENGES

As big of a overhaul as the mesh pipeline is, the transition from the traditional rendering pipeline to the a mesh pipeline introduces some challenges for developers intending to use them. These challenges are mostly originate from the flexibility enabled by the mesh pipeline, which also results in more responsibility for the developer. As such, this section explores the challenges these developers may face when transitioning from the traditional pipeline to the mesh pipeline.

3.1.1 *Dimensionality of the Task and Mesh Shaders*

In the traditional pipeline, every programmable shader has its frequency only defined in terms of geometry (*i.e.*, the vertex shader executes once per vertex, the geometry once per primitive). Task and Mesh shaders, just like Compute shaders, trust the developer to distribute the work across a number of thread groups (*i.e.*, work groups) of a certain size, with both the number of work groups to launch and their size being controlled by the developer (as well as the amount of work each does, obviously).

This choice is not to be made carelessly, as choosing the correct parameters is crucial in obtaining the best performance out of the mesh pipeline. Returning to the cube example of [Subsection 2.3.4](#), two approaches were used to render the cube. In this case, the model is too simple of a task for the GPU to handle, so both approaches result in similar performances. However, for more intensive geometric tasks and more distinct parameter values, this is not the case.

The flexibility of the mesh pipeline imposes on developers the job of optimizing the distribution of the task to the hardware. This contrasts with the traditional pipeline, which has a rigid structure and such optimizations already implemented. Thus, with the mesh pipeline, in addition to solving the task at hand, developers have to handle its distributed by the groups of threads. On top of that, the values for the parameters which produce the highest performance metrics are dependent both on the hardware and the task at hand.

To help with this task, [Chapter 4](#) presents several performance tests with varying parameters' values. These tests should allow for a better understanding of effects caused by changing each parameter in different scenarios. Nonetheless, these tests do not provide a definitive answer on which values should be used in every scenario.

3.1.2 Pre-processing of Data

As mentioned in [Section 2.3](#), Mesh and Task shaders can't use the usual [Vertex Buffer Objects \(VBOs\)](#) and [Vertex Array Objects \(VAOs\)](#) for storing the vertices' attributes. Instead, Mesh and Task shaders have to access other memory objects (e.g., [Shader Storage Buffer Objects \(SSBOs\)](#) or Textures) to retrieve not only the vertices attributes but also how they are combined to form primitives.

Thus, the developer is tasked with changing in its rendering software the process of conversion from the geometry definition files (e.g., Wavefront (.obj) files). Nevertheless, this process shouldn't be as trivial as replacing the current storage method and keeping a similar structure.

Mesh and Task shaders were designed to allow the developer to use their flexibility to achieve greater performance than previous implementations of the pipeline. As such, developers should have performance in mind when developing any part of a Mesh or Task shader, which includes the chosen data structure for the vertices and primitives.

By design, each Mesh shader work group should contribute to the final result a different set a primitives denominated by NVIDIA a *meshlet*. To achieve the best performance possible, each Mesh shader shouldn't fetch the *meshlet*'s primitives' information from disjoint memory addresses (e.g., the first primitive from the beginning of a buffer a second primitive from the end), in order to take advantage of the principle of locality present in the GPUs. When fetching information from memory, a chunk of data is cached, so that subsequent memory accesses for data within the chunk can be delivered faster. Thus, it's advantageous to have the *meshlet*'s data stored in a way that benefits the caching mechanism (e.g., with buffers the *meshlet*'s should be ordered sequentially).

In addition to this, it's also beneficial to have good vertex re-utilization within each *meshlet*. By re-utilizing vertices in several primitives, less memory accesses are performed, which results in better performance in the end.

Ergo, a developer who wants to take full advantage of the mesh pipeline's performance must pre-process each geometry file, in order to divide the geometry into a ordered set of *meshlets*, with each *meshlet* reusing the most amount of vertices possible.

To achieve this, Kubisch (2018) proposes three distinct buffers (shown in 19), which should work alongside the vertex buffer (*i.e.*, the buffer which contains the vertices). With this approach, the vertex buffer structure is kept as a list of unique vertices, but the primitive index buffer is replaced by three buffers: a vertex index buffer, a primitive index buffer and a *meshlet* description buffer. The vertex index buffer stores the indices of the vertices to be used in each *meshlet* sequentially. The primitive index buffer stores the primitives of each *meshlet* sequentially by referencing the indices stored in the vertex index buffer. The *meshlet* description buffer references sequentially the portions of the other two buffers that identify each *meshlet*. With this structure, the memory accesses done by each mesh work group should be lower, thus resulting in better performance.

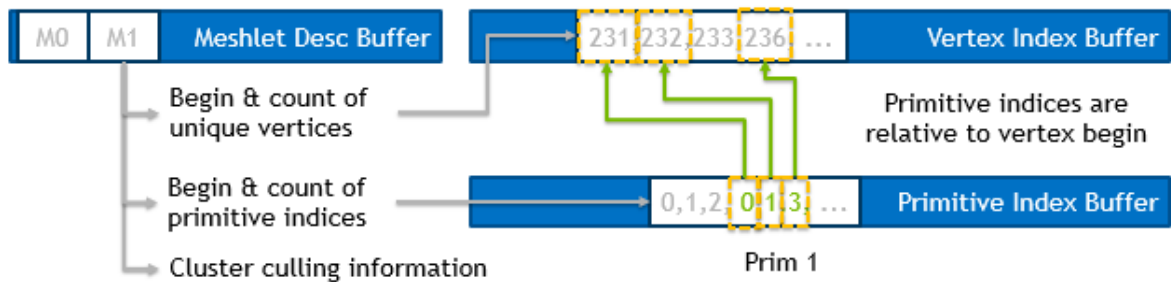


Figure 19: Proposed *meshlet* data structure (Kubisch, 2018).

With the storage of the *meshlets* sorted, the developer must explicitly in each Mesh shader access the *meshlet*'s data to emit its vertices and primitives.

To guide developers through this pre-processing of the data, Section 3.2 explains the development done throughout the thesis that enabled the rendering of 3D models with Task and Mesh shaders. Additionally, in Section 4.8, the performance obtained when rendering these models with Task and Mesh shaders is compared to the performance obtained when rendering them with the traditional pipeline.

3.2 RENDERING 3D MODELS

The next development step is to use Mesh and Task shaders to render 3D models (*e.g.*, Wavefront models). This will allow us to load higher geometry objects that are needed for meaningful performance comparisons between the pipelines. To do this, part of the challenge mentioned in Subsection 3.1.2 must be overcome, as the Nau engine provides partial compatibility for loading Wavefront files, but it does so exclusively to VAOs used by the Vertex shaders.

The solution to this problem is to use either a buffer type object or textures to pass the 3D model information to the Mesh and Task shaders. In this thesis OpenGL's SSBOs were chosen for this purpose.

To utilize said buffers, Nau allows the developer to specify a path to a file containing the buffer's data and choose a binding point for the buffer. The developer must also specify the dimensions of the file and its structure *e.g.*, a file with twelve lines where each line has three values, can be used for the primitives of the cube.

As such, a Lua script was written to retrieve the information from Wavefront files and translate it to several buffer files to be supplied to the Nau engine. Subsection 3.2.1 explains this translation script and covers the decisions made during its development.

With the 3D model information stored in the buffers, the Mesh shader must read from those buffers and render the model in a timely manner. Subsection 3.2.2 details this transition and provides illustrative examples of how the whole rendering process works.

3.2.1 Lua File Generator

To develop the buffer file generator, there were three main considerations to have: the input, the output and the conversion/translation algorithm. These three considerations are explored in detail in Subsubsection 3.2.1, Subsubsection 3.2.1 and Subsubsection 3.2.1, respectively.

Input

The first step in writing the buffer file generator is to retrieve the model information from the input files. Initially, a Lua module for Wavefront file parsing was obtained from a code repository but the former had a design flaw that prevented its use with larger models: the module read the entire model to a Lua table stored in memory, which hurt performance significantly.

Therefore, a new module was written from scratch to parse the file using iterators and also new functionality was added to support the material properties of the model. However, this parser still has some limitations:

- Supports as input exclusively Wavefront file types (*i.e.*, files with *.obj* as their extension);
- Only translates geometric vertices, texture vertices and vertex normals for the vertex data (*i.e.*, only lines starting with *v*, *vt* or *vn* are read);
- Only supports face elements for primitive description and its indices must be positive (*i.e.*, the indices can't refer elements starting from the bottom of the file);
- Although able to read material properties from material library files, only the diffuse color or the diffuse textures are retrieved at the moment.

Despite these limitations, the writing of this parser was paramount in enabling the thorough testing intended for this thesis, as it allowed a fast generation of buffers even for larger 3D models.

Output

With the parsing of the input resolved, the output must be determined. As mentioned in Subsection 3.1.2 and illustrated in Figure 19, NVIDIA suggests a three-buffered approach to feed the *meshlet* information to the Mesh shader.

This three-buffered approach covers solely the referencing of the vertices used in each *meshlet*, which is why extra buffers are needed to provide the Mesh shaders with the data of each vertex. As the generator grew in complexity and capabilities, the size and number of buffers also grew to accommodate the information needed to be passed on. To date, the generator creates the following buffers when parsing a Wavefront file:

1. A single buffer file for the vertices' coordinates of every vertex in the model.

2. A single buffer file for the vertices' normal coordinates of every vertex in the model, if the model contains them.
3. A single buffer file for the vertices' texture coordinates of every vertex in the model, if the model contains them.
4. Multiple vertex index buffers for every material in the model.
5. Multiple primitive index buffers for every material in the model.
6. Multiple *meshlet* description buffers for every material in the model.

In short, each Mesh shader invocation uses the same three "vertex properties" buffers and three buffers that change according to the material being rendered in said invocation. The multiple buffers for each material were used instead of three larger buffers for several reasons.

One of the reasons was to improve vertex locality, as vertices are more likely to be shared if they all use the same material. This is important as to avoid having to re-fetch vertices from memory that were already used.

A second reason was to avoid having the Mesh shaders inform the Fragment shaders of the type of material to render each primitive with. This means that the Fragment shader would have to choose from a set of material properties the ones referring to the material suggested by the Mesh shader for each primitive, which would increase the complexity of the Fragment shader. Instead, by using different buffers for different materials, one can easily nominate serviceable Fragment shaders for every material, depending on the needs of each one. As the purpose of this thesis is to compare the geometric part of the pipeline, it's in its best interest to minimize the Fragment shader's impact on the performance metrics recorded and designing an overly complicated Fragment shader could have proved detrimental to this goal.

The third and last reason was to avoid hitting the limitations of Mesh shaders' invocations imposed by the hardware. For example, in the testing hardware the maximum number of texture units a mesh shader can access in a single invocation is 32, meaning that a single shader can only retrieve information from 32 different textures. Workarounds could be found to circumvent these limitations but these are outside the scope of this work.

Besides these buffers, the buffer file generator also produces a file which lists the material properties retrieved from the material library and associates them with the other buffers created. This file is then used by a script which generates a Nau project to render the model with the correct materials linked to each Mesh shader invocation.

Algorithm

With both the input and the output sorted out, the actual conversion algorithm must be developed. There were two ways to approach this problem:

1. Create a simple translator to read the Wavefront file sequentially and fill the *meshlets* using the same ordering used in the original file.
2. Create a more complex converter that reads the entire Wavefront file and fills the *meshlets* according to the best overall vertex re-utilization ratio.

The chosen strategy was the first for several reasons:

1. Writing an algorithm of the complexity of the second approach would have been scope creep. There have been several complex articles written on optimizing the vertex ordering of a model to allow for better vertex reuse and, consequently, better cache locality. Although it would have been certainly beneficial to write or adapt an algorithm of said type designed for the *meshlet* structure, the complexity of this task would require significant research and an amount of time that would exceed the scope of this thesis.
2. The algorithm had to run in a performant manner for 3D models of varied vertex densities, because this translation algorithm would be used a significant amount of times, especially with the extensive testing done in this thesis. As such, the simpler approach results in a much less time consuming process which allows for time investments in either more testing or in the other aspects of this thesis.
3. The second algorithm would have to be crafted in a way to not create an unfair comparison between the mesh pipeline and the traditional pipeline. As both pipelines benefit from vertex reordering, one shouldn't have a better reordering algorithm than the other, as that would make pipeline performance comparisons biased towards the one with the better reordering algorithm.

As a consequence of this choice, [Section 4.3](#) covers how the various optimization algorithms developed by other researchers for the traditional pipeline fare when used to optimize a model that is used with the Mesh Pipeline. These algorithms are run before the translation process and the optimized ordering may also allow for a better *meshlet* creation.

With that said, the translator only needs to be aware of the Mesh shaders limitations to build each *meshlet*. As covered in [Subsection 3.1.1](#), Mesh shaders work groups are limited in the amount of vertices and primitives they can produce, hence *meshlets* must be sized accordingly. Thereby, the *meshlet* creation must be target to specific Mesh shaders work group configurations.

For instance, if the developer decides that the optimal configuration for the Mesh shaders is to have each work group produce X number of vertices and Y number of primitives, then the *meshlets* pre-generated should abide by those limits to avoid extra work during runtime.

Therefore, the translator algorithm iterates primitive by primitive, checking for each one if adding said primitive to the current *meshlet* doesn't push it above the vertex or primitive maximums. If adding the primitive surpasses the limits specified, a new *meshlet* is created and the previous is considered finished.

As mentioned in [Subsubsection 3.2.1](#), the translator shouldn't add primitives with different material properties to the same *meshlet*. Thus, the translator keeps in memory one unfinished *meshlet* for each material until every primitive has been processed.

For testing purposes, the ability to only partially translate a model by ignoring the model's normals and their material properties was also added to the translator. This allows the testing of how much the normals and the material properties affect the overall vertex reuse in the *meshlets*.

3.2.2 Updated Task and Mesh Shaders

With the buffers containing the model's properties generated as specified above, it's quite easy to code a Mesh shader capable of rendering the intended model. This Mesh shader can be divided into 4 simple steps:

1. Get the location of a *meshlet* to render using the `gl_WorkGroupID` (*i.e.*, use the thread group ID to access the *meshlet* buffer and retrieve the start location and number of both the vertex indices and primitives used by the *meshlet*);
2. Set the `gl_PrimitiveCountNV` to the number of primitives obtained;
3. Distribute the vertex definition by the threads in the group (*i.e.*, each thread uses its `gl_LocalInvocationID` to retrieve one or more vertices from the buffers and emits its position and other per vertex properties);
4. Distribute the primitive definition by the threads in the group (*i.e.*, each thread uses its `gl_LocalInvocationID` to retrieve the one or more primitives from the primitive buffer and emits the indices of the vertices that constitute the primitives).

These four steps are all it takes to create a Mesh shader which simply passes the model's information to the rasterization stage. These steps, although simple, still require quite more work than writing an equivalent Vertex shader.

3.3 TESSELLATION

In order to generate higher geometry objects for performance comparisons, one can use tessellation. As covered in 2.2.3, in the traditional pipeline there are shaders designed for that sole purpose, but with Mesh and Task shaders, it's the job of the developer to code a tessellation algorithm.

As such, this section will cover the work needed to implement tessellation with Mesh and Task shaders. However, the tessellation shaders present in the traditional pipeline are already matured and offer several types of tessellation and implementing every form of tessellation would be exhausting and outside the scope of this thesis. In 20, several types of fractional tessellation are shown with the same parameters (*i.e.*, two outer and eight inner tessellation levels), from the left to the right: triangle odd-spacing, quad odd-spacing, triangle even-spacing and quad even-spacing.

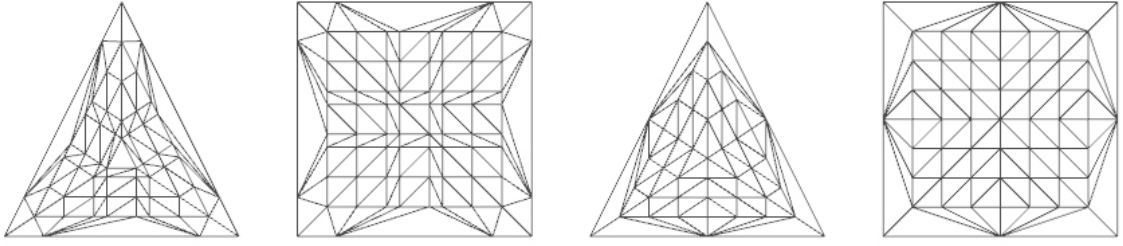


Figure 20: Some forms of Tessellation available with Tessellation shaders.

In this thesis it was explored the implementation of a single form of tessellation offered by the tessellation shaders: fractional even-spacing quad tessellation. This will allow to compare between the traditional pipeline featuring Tessellation shaders and the Task-Mesh shader approach. Although only a single form is implemented it should suffice for such a comparison, as all forms behave in a similar fashion.

During the development of this algorithm, there were several challenges that needed solving: some regarding tessellation itself and others regarding the adaptation to the Mesh shader paradigm. These challenges appeared as the complexity of the algorithm grew and more scenarios were tested.

To better demonstrate the challenges and how they were resolved, this chapter progresses roughly parallel to the development of the algorithm, with increasing algorithm complexity and problems needing solving. As such, the next subsections build on the work done by the previous ones and by the end a fully fledged tessellation algorithm is developed and ready for testing.

3.3.1 Non-Fractional Inner Tessellation

On the earliest of phases, the algorithm was only designed to take a quadrilateral and sub-divide into triangles given a non-fractional number of columns and rows. To do this, only one Mesh shader work group was launched to handle the tessellation of the quadrilateral.

This work group starts by calculating the number of vertices and primitives it needs to emit:

$$\#vertices = (\#columns + 1) * (\#rows + 1)$$

$$\#primitives = \#columns * \#rows * 2$$

With the number of vertices counted, the work group can distribute the calculation of vertices' positions by its threads. Each local thread uses its identifier to find the positions in the vertex grid it must generate vertices for.

In the context of tessellation shaders, generating a vertex means finding its tessellation coordinates¹. With the tessellation coordinates, the developer is tasked with deriving the properties of the vertex (e.g., spatial coordinates, normal, texture coordinates). This is usually done by interpolation of the original vertices' properties.

To emit the vertices generated, these must be stored in an one-dimensional array. The order in which the vertices are stored is relevant, as the primitives will be emitted by referencing the position of the vertices in this

¹ For a quad, the tessellation coordinates are a pair of normalized coordinates, which represent a position in a unit square with (0,0), (0,1), (1,0) and (1,1) as the coordinates of its vertices

array. As such, the vertices are stored in the array by row, meaning that a vertex situated in column zero and row one has as index the number of columns chosen for tessellation.

With the vertices emitted, the primitive indices must also be distributed across the threads. This is done by primitive, with each thread emitting one or more primitives (*i.e.*, multiples of three indices, as primitives are triangles in this case). With that said, distributing the primitives by the threads was done a similar manner as the vertices, using the local thread identifier to assign the primitives. Constructing the primitives, in this stage, is a trivial manner and depends only in the order chosen when emitting the vertices.

Emitted all the primitives and set the primitive count, the result of running this algorithm can already be compared side by side to the output of a Tessellation shader in the same scenario. For instance, for a 14x14 grid scenario, [Figure 21](#) pictures the result provided by the Tessellation shaders (shown on the left in black) and the one produced by our algorithm (shown on the right in red).

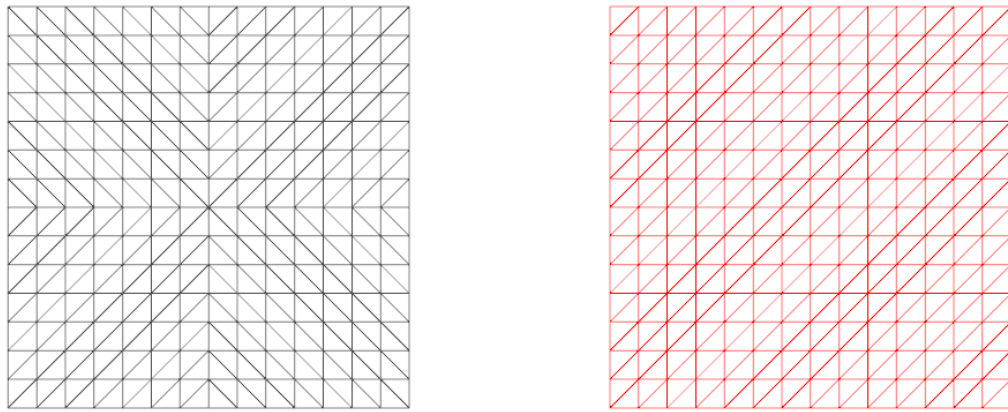


Figure 21: Non-fractional Inner Tessellation (14x14).

Right off the bat, the grids are not structured exactly the same, with the tessellation shaders rotating the triangles depending on their quadrant. Our algorithm doesn't do that, as it wasn't deemed necessary, even though it would be a simple fix.

Additionally, this example shows the most tessellation levels the algorithm could take and still produce the same result as the traditional pipeline (assuming an equal number of columns and rows). This was due to the output limitations of a single Mesh shader work group in the testing hardware: 256 vertices and 512 primitives. Advancing to the next level of tessellation with even spacing, the sixteen columns and rows would need to produce 289 vertices, as per the formula above, which goes over the limitations. This forces the developer to launch more than one work group if the intention is to at least match the sixty-four levels the tessellation shaders can output.

To fix this problem, a possible solution could be to just launch a number of work groups high enough to cover the vertex output needs of the task at hand. For instance, if the maximum amount of tessellation needed by the developer is 15x15, then the developer could set the number of work groups to launch to 2, because two work

groups are enough to cover the highest tessellation scenario. However, this would be inefficient, because two work groups would be used regardless of the chosen tessellation level, when for tessellation levels up to a 14x14 grid a single work group would suffice.

As such, a cleaner solution is to dynamically launch the Mesh shader work groups based on the demands of the task. To do this, a Task shader can be used to calculate appropriate number of work groups to launch using the number of vertices needed to achieve the tessellation levels intended. For static levels of tessellation, this calculation could be done on the CPU a single time for each tessellation level used and consulted at will, but to allow for dynamic levels of tessellation the Task shader was used.

This calculation, however, isn't as easy as dividing the total number of vertices needed by the maximum number of vertices the Mesh shaders can emit (considering that some of the vertices must be repeated with each division). To take advantage of vertex locality, the vertices should be divided in ways that benefit their reuse, so that in higher geometry cases work groups don't end up emitting geometry in a strip manner.

Thus, the Task shader divides the quadrilateral into other quadrilaterals to preserve the grid formation of the vertices, which may benefit from vertex reuse in up to six primitives. In comparison, in a strip a vertex can be shared only up to two primitives. Consequently, using a grid formation should, in theory, allows work groups to produce more primitives for the same amount of vertices than a strip like formation would.

Figure 22 provides a simple illustration of this theory for two tessellation levels (*i.e.*, 5x5 and 6x6 grid), where it's assumed that the Mesh shader work group can produce only nine vertices. Figure 22a and Figure 22d represent the case for strips, where a work group emits a strip of a fixed size. Figure 22b and Figure 22e represent wrapped strips, which improve the latter case by allowing a work group to emit every vertex it can, by wrapping the strip to the following row. Figure 22c and Figure 22f represent the quadrilateral approach, where work groups emit quadrilaterals of a predefined size without any morphing.

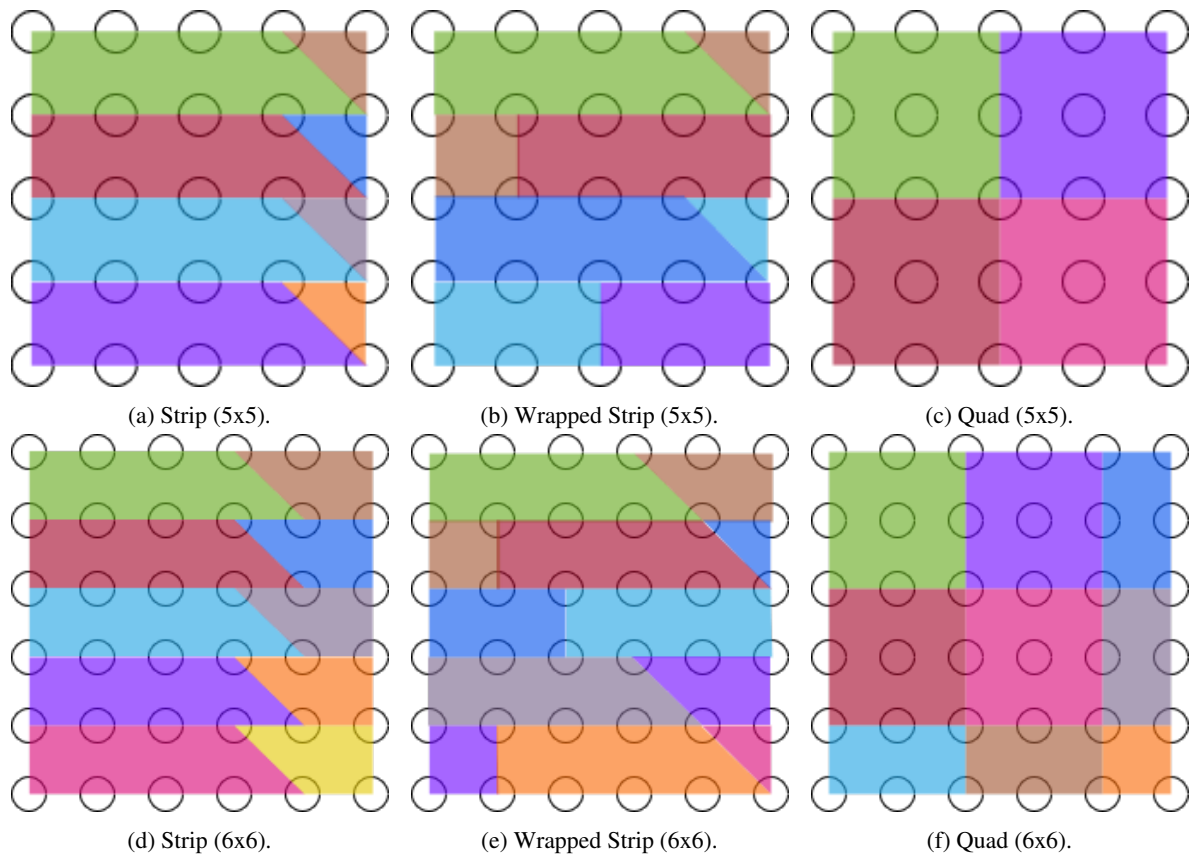


Figure 22: Number of Mesh shader work groups launched based on their configuration.

In terms of complexity, both the strip and the quadrilateral approach are straightforward to implement, as each work group can easily calculate the borders of their operation. The wrapped strip, however, requires each work group to account for the wrapping of previous strips in the calculation of their initial vertices and for its own wrap to calculate their last vertices. Regardless of complexity, the wrapped strip couldn't launch fewer work groups than the quadrilateral approach, with the quadrilateral approach emitting two less work groups in its best scenario and the same number of work groups in its worst.

Nevertheless, [Figure 22](#) illustrates only two tessellation examples and consequently doesn't paint the full picture. The higher the tessellation grows, the more shared vertices are shared by the quadrilateral approach in comparison with the strip approaches. With a work group that can emit more vertices, a strip approach may encompass several lines, upping the vertex reuse to a quadrilateral standard. However, work groups can produce a limited amount of vertices and thus, with higher geometry, the case of strips encompassing several lines doesn't really happen.

So, is the quadrilateral approach the best way to achieve the tessellation level desired with as few work groups as possible? Not quite. The quadrilateral approach fails in the cases where the edge work groups aren't producing as much vertices as possible. If a considerable amount of work groups aren't producing as much as possible, the benefits from the vertex reused gained by going with the quadrilateral approach can be outdone by other approaches that fit better at the edges.

An improvement to the original quadrilateral approach would be to resize the quadrilaterals to not have to strictly produce squared quadrilaterals but be able to produce quadrilaterals with different numbers of columns

and rows. Figure 23b proves that, in certain cases, resizing the quadrilateral may result in less work groups launched. However, one can see that resizing the quadrilateral from a square to a rectangle reduces the vertex reuse and is only better in specific cases. In this particular example, Figure 23c represents an optimal approach in terms of work groups launched as only Figure 23d can launch only six work groups and produce the overall grid. Among the two optimal approaches, Figure 23d can be considered better considering it emits less vertices overall.

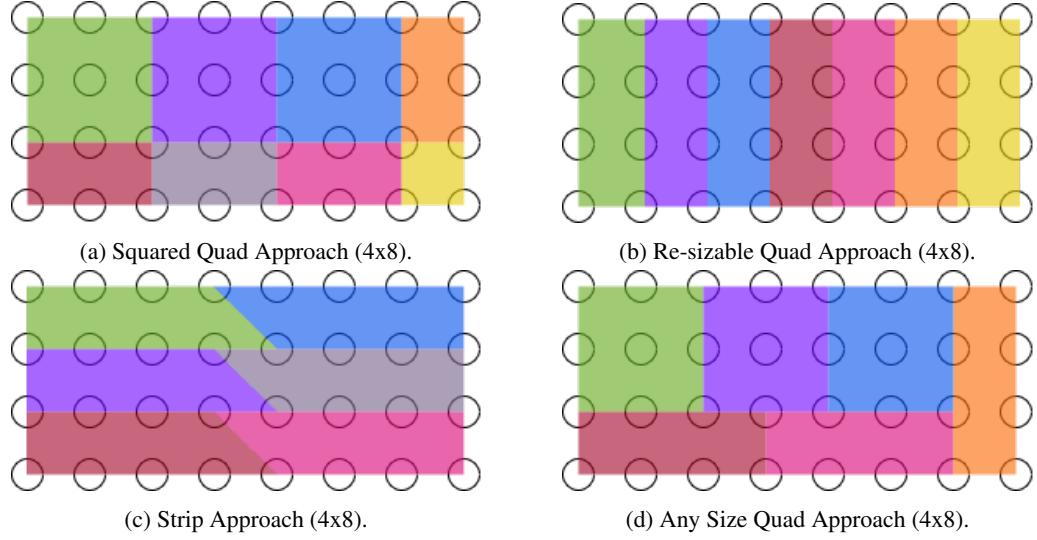


Figure 23: Number of Mesh shader work groups launched based on their configuration.

With the strip approach put aside for worse scalability, Figure 23 presents three approaches based on quadrilaterals with different levels of success and complexity. When using simplest approach, illustrated in Figure 23a, the number of Mesh shader work groups can be calculated with Equation 3, which multiplies the number of work groups needed to emit a single row (calculated with Equation 1) by the number of work groups needed to emit a single column (calculated with Equation 2).

$$\#col_divs = \text{ceiling} \left(\frac{\#cols}{\text{floor}(\sqrt{\#max_verts}) - 1} \right) \quad (1)$$

$$\#row_divs = \text{ceiling} \left(\frac{\#rows}{\text{floor}(\sqrt{\#max_verts}) - 1} \right) \quad (2)$$

$$\#work_groups = \#col_divs * \#row_divs \quad (3)$$

In Equation 1, the number of work groups needed to emit a row is calculated by dividing the total number of columns by the square root of the maximum vertices a work group can produce, because the approach shown in Figure 23a assumes every work group generates square grid. With the approach shown in Equation 2, however, the work group can produce any type of rectangle, which introduces more variables to the calculation and turns the problem into a discrete optimization problem. In this discrete optimization problem, Equation 1 and Equation 2

bare no significance and the goal is to minimize Equation 3, following the restrictions of Equation 4, Equation 5 and Equation 6.

$$\text{ceiling} \left(\frac{\#cols}{\#col_divs} + 1 \right) * \text{ceiling} \left(\frac{\#rows}{\#row_divs} + 1 \right) \leq \#max_verts \quad (4)$$

$$1 \leq \#col_divs \leq \#max_verts/2 - 1 \quad (5)$$

$$1 \leq \#row_divs \leq \#max_verts/2 - 1 \quad (6)$$

To find the answer to this problem, one can attempt to solve it mathematically as if its variables weren't discrete (*i.e.*, integer values) and then round the final result to a discrete variable. However, solving the problem this way, doesn't produce the outcome desired as can be exemplified with Figure 23. Figure 23a shows the need of three and a half work groups by row and one and a half by column, which results in five and a quarter work groups. Without the discrete portion of the problem, this is a better result than the seven work groups shown in Figure 23b, but if we round up the number of divisions by column and row, it no longer is.

Thus, this problem can't be solved by this method and another must be chosen. The chosen method was brute force (*i.e.*, trying all possible combinations), which usually isn't a great approach. However, due to the limitations of Mesh shaders, it's not that expensive computationally. In the test machine, Mesh shaders are limited to a maximum of 256 vertices, which corresponds to a grid of 15 columns and 15 rows. Thus, the combinations with 1 to 15 columns – considering for each as much rows as permitted – and with 1 to 15 rows – with as much columns as possible – must be tested, which results in 30 possible combinations to test. Of these 30 combinations, the 15x15 has the most vertex reuse, consequently being the most used. With these combinations, the bigger the difference from the number of columns and rows chosen, the more strip-like the approach becomes and the least likely it is to be chosen.

However, if the Mesh shader is set to output a maximum of 64 vertices instead, then only 14 combinations will be computed. Essentially, the number of combinations needed to be considered for a given maximum amount of vertices the Mesh shader can output is $2(\sqrt{\#max_verts} - 1)$.

As each of these combinations requires the same computation as the single calculation done for the approach of Figure 23a, the approach of Figure 23b is as much more computational expensive as the number of combinations tested, when comparing only the work done by the Task shader. But does this added computation in the Task shader compensate for the possibly higher number of work groups launched with the previous approach?

To test this, a Mesh shader was written to distribute the grid based on the divisions provided by the Task shader. In other words, the Mesh shader will task each work group with drawing a quadrilateral with a number of columns and rows inversely proportional to the number of column and row divisions calculated by the Task shader.

With this Mesh shader, we can set different numbers of divisions in the Task shader and see the performance impact of using more work groups. Section 4.4 covers the tests made to measure this impact and also covers the tests made to compare both the approaches of Figure 23a and Figure 23b.

Section 4.4 concludes that the benefits of the re-sizable approach don't always compensate for the higher computation done in the Task shader. However, when it does, it is more impactful to the performance in a

positive way than it is in a negative way in the other cases. Of course, being more or less impactful also depends on the frequency in which each case occurs, and that can only be measured when a given task is known.

As for the approach presented in [Figure 23d](#), it requires both Task and Mesh shaders to be more complex and figure a combination of possibly differently sized quadrilaterals to handle the vertices in the edge of grid. Because of the higher complexity of this approach and the broader scope of this thesis, it was decided to leave the evaluation of this approach for future work.

3.3.2 Fractional & Outer Tessellation

With all the work done in [Subsection 3.3.1](#), one can mimic the behaviour of the inner tessellation of the Tessellation shaders for quads, when equal spacing is set in the Tessellation Evaluation shader. However, the goal is to mimic even spacing method in order to be able to use fractional levels of tessellation.

Fractional levels of tessellation allow for smooth transitions between different levels of tessellation, removing the [popping](#) effect that occurs with the equal spacing method. Being able to specify fractional levels of detail is vital for many of the uses for tessellation and adding such functionality is straightforward, as long as there are no outer levels to worry about.

As a practical example, picture a grid with three columns and a grid with four columns. In each grid, the vertices of each row are equidistant from each other and a row from the grid with four columns has five vertices, whilst the row from the grid of three columns has four. This means that when smoothly increasing the number of columns from three to four, the four equidistant vertices of each row must morph into five equidistant ones.

To achieve smoothness in this process, the first step is to avoid having the vertex pop into existence in the row. To do this, the new vertex can originate from the position of a previous vertex. The second step is to linearly interpolate the positions of the vertices according to the tessellation level provided. Returning to the example above, if the desired tessellation levels for the columns is 3.1, then the position of the vertices must be way closer to original position than the final position.

With this in mind, one must choose a strategy for the appearance of the vertices. Naively, one can make the new vertices appear from the origin of the grid (e.g., the new column that originates would be the first column in its grid).

When translating this strategy to Mesh shaders, the previous method for calculating the position of a vertex must be changed. For each vertex, each Mesh shader calculates start and final positions for it based on the encompassing non-fractional tessellation levels, and linearly interpolates between them using the fractional tessellation level. [Figure 23](#) shows this process for a row and different tessellation levels. The figure also pictures the indices of each vertex that are used to figure out the starting positions of each (i.e., when $n = 3$) and their final positions (i.e., when $n = 4$).

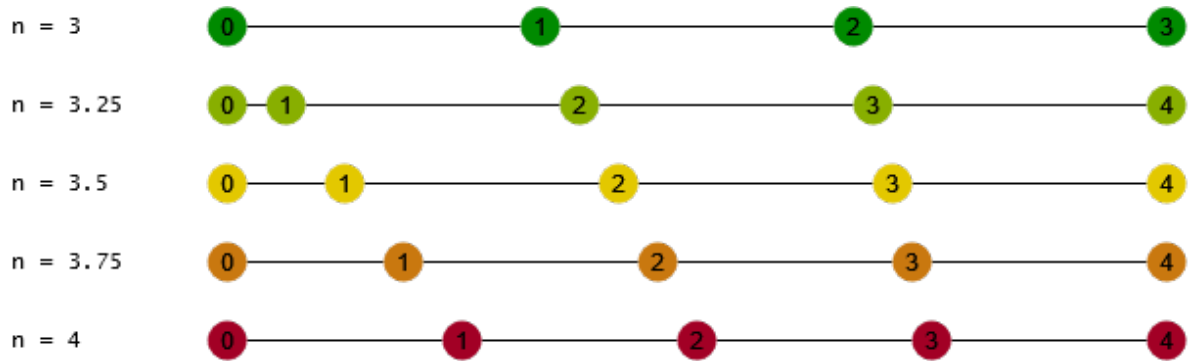


Figure 24: Transition between two levels of tessellation..

This strategy is valid but when adding outer levels to the tessellation, it introduces some problems. The outer levels of tessellation refer to the number of divisions the edges of the quad has. Having different levels of tessellation for the edges of a quad, allows quads with different levels of detail (*i.e.*, different inner tessellation levels) to be stitched together without noticeable "holes" by having the same number of outer tessellation levels.

Having outer tessellation levels complicates the grid because a strategy for the primitive generation at the edges must be chosen. To help describe possible strategies, the term *base* is used to describe the side of the triangle composed by the two vertices of the same row and the term *tip* to describe the other vertex. A triangle may also be referred as a triangle from a specific row to specify a triangle whose base is in said row.

A simple strategy, to solve the problem of creating a triangle strip with different number of vertices on each row, is to consider the first vertex of the bottom row the tip for every triangle in the top row and the last vertex of the top row the tip for every triangle in the bottom row. This strategy effectively creates a strip but it does so with very skewed triangles. Figure 25 illustrates this strategy on a simple configuration and one can see that some triangles have very elongated edges, whilst others don't.

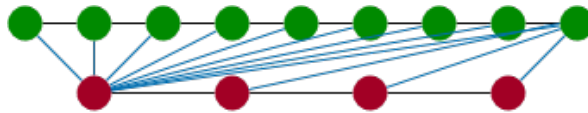


Figure 25: Naive approach to outer primitive construction.

As such, a strategy must be found to better connect the vertices and create triangles with a smaller footprint (*i.e.*, area). A different strategy could also be to connect the base to the closest tip. This would result in configurations such as the ones shown in Figure 26. In each of those configurations, the result is as intended, regardless of the chosen levels.

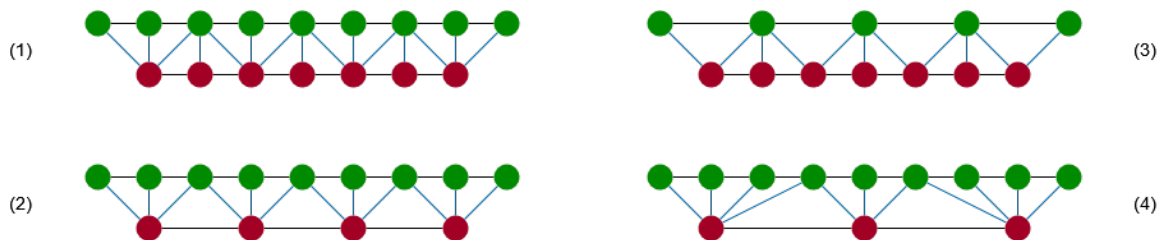


Figure 26: Primitives on the outer levels.

However, choosing the tip based on this method introduces a problem. Following the goal of maintaining a smooth transition in between tessellation levels, if one chooses the previously mentioned approach of introducing vertices through the origin point, the bases will consistently change their tip as new vertices are introduced. This change is abrupt, as there's a point in which another vertex becomes closer to the base and the tip is switched in a single frame.

Consequently, a requirement to guaranteeing smoothness when transitioning between tessellation levels, is that the previous connections between bases and tips must be maintained. Because of this requirement, the strategy of introducing a vertex always in the same location can't be used, because maintaining the previous connections results in the skewing of triangles as the tips become more distant from the bases.

In order to solve this conundrum, one solution is to copy the strategy used by tessellation shaders. After a bit of reverse engineering, the tessellation shaders seem to be using exponents of two to solve both the tip matching and the introduction of new vertices.

The first thing the tessellation shaders do is separate the fractional tessellation into two modes: even or odd. This means that a developer using fractional odd spacing can't get a grid with an even number of rows or columns. For example, if a developer using fractional odd spacing specifies 4 tessellation levels for the columns, the tessellation shaders interpret the 4 as the middle ground between 3 and 5 columns. Consequently, with each level change (*i.e.*, each two levels) two vertices are added.

Figure 27 displays the process of increasing the outer levels (using even spacing) between two exponents of 2 (*i.e.*, 2^3 and 2^4). Blue dots represent the two most recent vertices to be added and the yellow ones the vertices from where these originated.

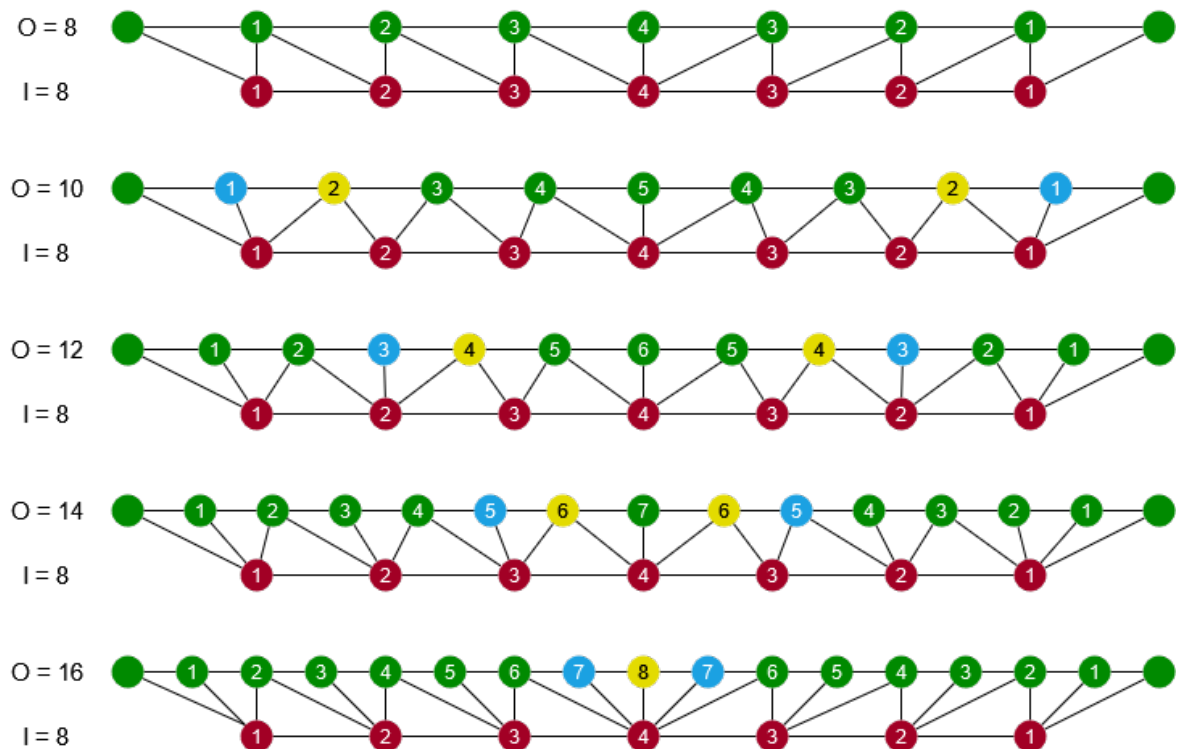


Figure 27: Increasing outer levels in the Tessellation shaders.

As can be seen from Figure 27, initially the vertices originate from the edge, but with each step they start originating from vertices closer to the center. Essentially, when increasing the tessellation from one exponent of two to the next one, every vertex, excluding the edge ones if in the outer row, is used as the origin point for new vertices, following an inwards order.

Translating to code, one can calculate from which of the original vertices the new vertex must originate from based on the tessellation by using the logarithm of base two:

$$idx = \text{ceiling} \left(\frac{tess - \text{floor}(\log_2 tess)}{2} \right) \quad (7)$$

For example, for a tessellation level of 13, Equation 7 calculates that the new vertex originates from the vertex number 3. If we look at configuration $O = 8$ in Figure 27, this means that for $O = 13$ the new vertices must originate from the green dots numbered with a 3.

Nonetheless, when looking at configuration $O = 14$, the yellow vertices are numbered with a 6. This is because 3 vertices have already been introduced and so the original vertex number 3 has become number 6.

With this strategy vertices are introduced evenly throughout the rows, which allows previous triangles to stay connected without getting increasingly elongated.

However, introducing the vertices in this manner isn't the full story to this tessellation method. Although it might seem like it, the bases of the triangles aren't connected to the closest vertex of the opposite row. Instead, the tessellation shaders connect the bases to the tips according to exponents of two, as well. To help understand how the base to tip connections are made, the word ratio will be used to describe how many consecutive bases connect to the same opposite tip. This ratio always to the bases of the highest tessellated line (*i.e.*, with a higher tessellation level).

For example, it can be seen in Figure 27 that when both levels are set to the same exponent of two, each base connects to a different tip (*i.e.*, a ratio of 1), except for the bases at the center, which always connect to the same tip due to symmetry. But when one of the levels is one exponent higher, it becomes two bases for each different tip (*i.e.*, a ratio of 2).

The story complicates a bit when the levels aren't set to exponents of two. In configuration $O = 12$, starting from the edges, the ratio is 2 for every base before the yellow vertex and 1 for every base following it.

Figure 28 takes this specific case and increases the inner tessellation to 10. This results in the most complex case that can be manufactured, where there are two different ratios, with one ratio interspersing the other. In this case, we got 1 base per tip before the appearance of an opposite yellow vertex, then 2 bases per tip until the adjacent yellow vertex and 1 base per tip for every following base.

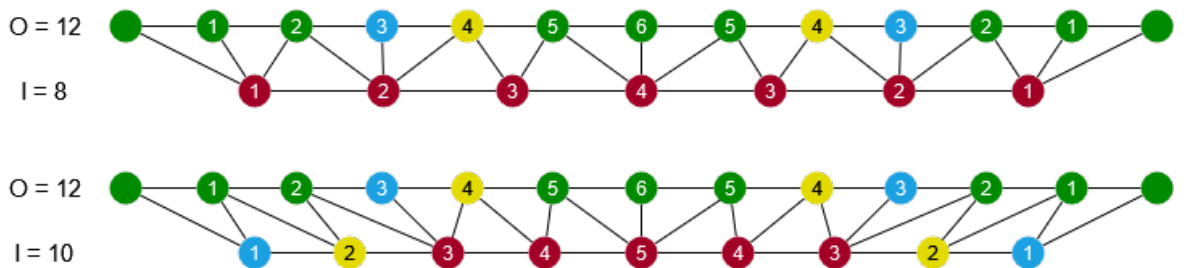


Figure 28: When levels aren't powers of two.

With both the strategy for introducing new vertices and connecting bases to tips figured out, all that's left to do is translating them into the Mesh and Task shaders, in order to have our tessellation algorithm support both fractional tessellation and outer tessellation levels.

As mentioned before, the fractional part of the tessellation can be implemented for each vertex by linearly interpolating between a start position and a final position. For example, given a tessellation level of 12.5, the start position of every vertex is calculated for tessellation level 12 and the final position for tessellation level 14. With tessellation level 12 there's one less vertex than with 12.5, so every vertex until the newly introduced vertex, inclusively, uses its current index to calculate the start position and every vertex after it uses its index decremented by one. This results in the new vertex (shown in blue in Figure 27) and the one following it (shown in yellow) having the same origin position, just as intended.

With the start and end positions calculated, as 12.5 is 25% between 12 and 14, one can calculate the actual position by interpolating the start and end position using the 25%. As a concrete example, for vertex number 1 we got a start position of $1/12 \approx 0.083$ and end position of $1/14 \approx 0.071$. Consequently, for a tessellation of 12.5 the horizontal tessellation coordinate for vertex number 1 is $0.083 + 0.25 * (0.071 - 0.083) = 0.080$.

As for the outer tessellation, implementing it requires a bit more thought. Because the outer tessellation is quite different from the inner tessellation, an entirely different set of shaders (*i.e.*, a Task and a Mesh shader) was written for it.

To simplify the process, each Mesh work group is assigned with a single side of the quadrilateral. Consequently, at least four Mesh work groups must be launched to handle a quadrilateral, even if its outer levels could be emitted by fewer. When a side's tessellation is high enough to require more than one Mesh work group, the primitives must be distributed across several work groups.

Distributing the primitives across the work groups functions entirely different from the way it's done for the inner tessellation. With outer tessellation, there might be a different number of vertices in each row and, depending on the base to tip ratio, one vertex of a row may be useful to a number of primitives, instead of just two.

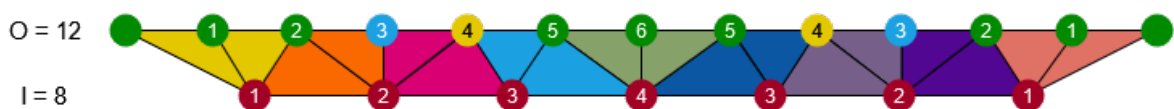


Figure 29: Example of optimal distribution of the primitives by the mesh work groups.

For instance, Figure 29 pictures how primitives could be optimally distributed by mesh work groups, limited to a maximum amount of vertices of 4. In this figure, each color signifies a different mesh work group in charge of emitting said colored primitive (*e.g.*, the yellow colored primitives are emitted by a different work group than the orange ones). However, to construct this scenario, the primitives were filled with colors from left to right, thus having known for each consequent work group the primitives and vertices already assigned to previous work groups.

This creates a problem because, even though it would be easy for the Task shader to calculate the minimum amount of work groups necessary to emit the strip, each Mesh shader work group has to calculate where its vertices and primitives start and end. For example, the first set of primitives has 3 outer vertices and 1 inner vertex, whilst the subsequent groups has 2 two of each.

It is possible for each Mesh work group to calculate exactly its boundaries for this scenario by simulating the distribution the primitives across the work groups, using the several ratios in conjunction with the positions of the new vertices, just like it's done to calculate the tips for a specific base.

To avoid all this calculation done just so each work group knows which vertices it has to emit, a simpler distribution algorithm was used. Instead of distributing the primitives sequentially, the bases of the highest tessellated side are distributed according to the minimum base to tip ratio of each scenario. An example of how this distribution is done is shown in Figure 30, for the same scenario presented in Figure 29.

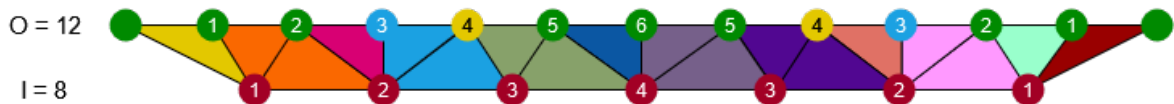


Figure 30: Example of the distribution method used.

For the scenario pictured, the minimum base to tip ratio is 1 and the maximum amount of vertices a work group can emit is 4. With these values, the algorithm distributes the bases of the highest tessellated side assuming that for each base a unique tip is necessary. This results in each work group being assigned a base, due to the limit of 4 vertices (*i.e.*, if each was able to emit 5 vertices, then two bases could be assigned to each work group).

With this distribution algorithm, is trivial to distribute the primitives as each work group emits exactly the same amount of bases in the highest tessellated side, as can better be seen in Figure 31. Each Mesh work group calculates the tip for it's first and last base (in this scenario, as each work group was assigned a single base, the first and last base are the same), thus getting the range of the strip it has to build.



Figure 31: Distribution of the bases by the work groups.

However, by calculating the tips for the first and last bases, there might still be primitives left to emit, pictured in Figure 31 as colorless primitives. To solve this problem, instead of each Mesh shader calculating the tip for its last base to determine the end of its range, it calculates the tip for the base which follows the last base. As this might result in an extra vertex for work group, this vertex has to be accounted for in the Task shader.

Returning to the scenario of Figure 30, the range of red vertices of the yellow work group ends in 1, because the 1 is the both the tip for its base of green vertices, as well as the tip for the subsequent base. However, for the orange work group, the range of red vertices ends in 2, solely because the next base's tip is number 2. As a consequence of this, the uncolored primitive which sits in between the orange and magenta primitives in Figure 31 was assigned to the orange work group.

The trade-off of this simpler approach is the launch of more mesh work groups than absolutely necessary, as can be seen by comparing Figure 29 and Figure 30.

In the end, mimicking the behaviour of the Tessellation shaders' even spacing fractional tessellation is not a trivial task, especially if good performance is a requirement. In this section the approaches chosen to mimic this behaviour weren't always the most performant amongst the options. For example, to mimic the full behaviour

of Tessellation shaders, two launch calls are launched: one to emit a Task shader to handle inner tessellation and one to handle the outer one. It would be possible to combine the Task and Mesh shaders into a single call, which would most likely help the performance, but to avoid the development of unnecessarily complex Task and Mesh shaders, the two launch calls were maintained. This philosophy should be taken in consideration when observing the performance results obtained in [Section 4.9](#).

3.4 SUMMARY

This development presented in this chapter should be enough to show that developing with Mesh and Task shaders requires additional considerations not needed with the traditional pipeline. These shaders fully succeed at bringing the compute paradigm to the rendering pipeline, but do so at the cost of convenience.

CASE STUDIES / EXPERIMENTS

In this chapter, the developments made in [Chapter 3](#) will allow for a broad analysis of the performance that can be obtained when using the mesh pipeline instead of the traditional one.

In the first part of this chapter (*i.e.*, from [Section 4.3](#) to [Section 4.7](#)), several approaches of utilizing Task and Mesh shaders will be explored, in order to decipher the conditions in which they perform at their best.

In the second part of this chapter (*i.e.*, from [Section 4.8](#) to [Section 4.10](#)), these best conditions will be put to the test against the traditional pipeline to find which of the two achieves the best performance in the different scenarios.

For some of these experiments, 3D models were used. The models used were retrieved either from the McGuire Computer Graphics Archive ([McGuire, 2017](#)) or the The Stanford 3D Scanning Repository ([Stanford, 2014](#)).

4.1 MEASURING PERFORMANCE

For measuring the performance of a pipeline, the Nau 3D engine has a built-in profiling menu that allows the examination of the time each step took in the rendering of a frame. However, this menu is meant for real-time debugging purposes and not for the collection of performance data for later evaluation.

As such, a second functionality of this engine had to be used for our tests: Lua scripting. Nau allows Lua functions to be executed right before and right after the execution of a pipeline and these functions can have access to the information the engine gathers. For our purposes, we intended to retrieve both the time it took to execute the pipeline and the number of frames rendered during that time, and then output those values to a [Comma-separated Values \(CSV\)](#) file for further processing.

Therefore, the performance metrics obtained for this thesis were always retrieved via a two Lua functions: a `startTimer` function, executed right before the execution the pipeline, and `stopTimer` function, executed right after it. Depending on the pipelines and tests executed, different variations of the `startTimer` function were executed in order to output more specific information about the pipeline, such as the type (*i.e.*, traditional or mesh) and the parameters used (*e.g.*, `max_vertices` or `max_primitives`).

Regarding the hardware, the NVIDIA GeForce RTX 2060 Mobile graphics card was the only graphics card used throughout the development of this thesis. The RTX 2060 Mobile is a first generation GPU when it comes to Mesh and Task shaders, which means it's one of the first hardware packages to support said functionality.

4.2 STANDARD ERROR

The performance metrics obtained throughout this thesis can't be immediately taken as face value. Although the testing methodology isn't flawed in and of itself, it could be improved in many ways. Because these tests were run on a Windows laptop, it's hard to maintain a controlled environment in which to run the tests.

To help mitigate the variations on the tests due to external factors, some procedures were taken before the execution of the tests:

1. All the software not necessary for the execution of the tests was stopped to avoid background tasks interfering with the results obtained. Nonetheless, some [Operating System \(OS\)](#)-specific tools can be exceptionally hard to turn off (e.g., Windows Defender) and could still have impacted performance.
2. All the tests designed to be compared amongst each other, were ran in sequence and in a random order. By running them in a sequence we guarantee minimum interference to the environment in which they are ran and by sorting them arbitrarily we eliminate the possibility of taking conclusions about parameters only because they were ordered in a certain way.
3. When possible, the same test was executed several times in order to eliminate outlier results.

Despite the measures taken, there are still external factors that can affect the measurements of performance taken during the tests. One of these factors is the frequency scaling that is done to the hardware to accommodate for the fluctuations in temperature (*i.e.*, thermal throttling).

Consequently, there are still noticeable variations when executing the same test several times. As such, a standard error must be calculated to establish the margin of error of our results. To do this, different types of work loads for mesh shaders were tested using tessellation. Each tessellation level tested was executed 20 times in alternating order. With said measurements, the standard error was calculated, as well as the margin of error for a few confidence intervals. The results of these calculations can be seen in [Table 1](#).

Tess. Lvl	Mean	SD	SE	ME (90%)	ME (95%)	ME (99%)
256	836,28	27,83	6,22	±10,76	±13,03	±17,81
512	840,53	30,90	6,91	±11,95	±14,46	±19,77
1024	673,73	32,85	7,35	±12,70	±15,37	±21,02
1536	373,99	11,31	2,53	±4,37	±5,29	±7,23
2048	312,68	9,62	2,15	±3,72	±4,50	±6,15
2560	254,47	11,95	2,67	±4,62	±5,59	±7,64
3072	197,44	4,15	0,93	±1,61	±1,94	±2,66
3584	160,82	3,28	0,73	±1,27	±1,54	±2,10

Table 1: Margin of Error of the Performance Metrics ([frames per second \(FPS\)](#)).

^{SD} Standard Deviation

^{SE} Standard Error

^{ME} Margin of Error

The results achieved with this calculation corroborate that measuring the performance of tests with non-demanding workloads - and consequently high FPS - can result in metrics which vary significantly from test to test. This erratic behaviour can even be seen within this test designed to measure said behaviour.

By comparing the results of the lowest tessellation levels (*i.e.*, 256 and 512), one can observe that the 512 tessellation level had higher average FPS recorded than 256, even though it rendered almost 4 times more triangles. Ordinarily, the higher workload would result in lower performance overall.

However, upon further inspection, the difference in average FPS falls within the margin of error calculated with this test and, as such, one can't accurately confirm performance differences between the two, because obviously there are external factors to the geometric pipeline at play affecting the performance measured.

Thus, throughout this thesis the margins of error calculated in this section should be taken into account to distinguish which performance differences are considered significant and reproducible.

4.3 VERTEX LOCALITY OPTIMIZERS

When rendering 3D models with Mesh shaders, a developer must get hands-on with the transformation of the meshes into *meshlets* (*i.e.*, smaller meshes that can be rendered by a single Mesh shader work group). While doing so, the developer shouldn't be naive and think that the organization chosen has no impact in the final performance of the Mesh shader pipeline.

As seen in Section 4.4, trying to minimize the number of work groups launched results in better performance in the end. Thus, when rendering 3D models the developer must have this goal in mind as well.

As work groups have a maximum output number of vertices, to minimize the number of work groups launched the developer must maximize the number of primitives launched by each one by reusing the greatest possible amount of vertices. This type of optimization has been in use with the traditional pipeline for years and several algorithms have been researched for this purpose.

In this section, we explore the impact of pre-processing a 3D model with the algorithms of Hoppe (1999) and Forsyth (2006). To pre-process the models with these algorithms, the *DirectXMesh* tool was used.

As there's a possibility that the original model is already nicely optimized, a script was written to shuffle the lines of a Wavefront model which refer to the primitive indexes. Shuffling said lines effectively removes any previous vertex locality optimization and results in a model where these algorithms can have a true impact.

With that said, six test scenarios are presented:

1. Original model with no optimization algorithm applied
2. Original model with Hoppe's optimization applied
3. Original model with Forsyth's optimization applied
4. Randomized model with no optimization algorithm applied
5. Randomized model with Hoppe's optimization applied
6. Randomized model with Forsyth's optimization applied

For each of these scenarios, the traditional pipeline was tested as well as four differently configured Mesh shaders. Despite the presence of both the traditional pipeline and some configurations for Mesh shaders, these are not to be compared amongst each other but with themselves across the several scenarios.

The model where the vertex locality optimizer algorithms were tested was the PowerPlant model. Table 2 presents the FPS recorded with the different scenarios for said model in different pipelines, with the best performance for each pipeline being shown in bold. One can see that the traditional pipeline benefited from the optimizations, but the differences in performance were way more notable with the randomized model. When we observe the Mesh shader configurations, the story changes a little. Whilst with the randomized model, the optimizations provoked unequivocally better performance, with the original model the results don't match the ones of the traditional pipeline. In only one of the configurations did an optimized model perform better than its non-optimized version. This may indicate that these algorithms perform more consistent results with the traditional pipeline, which they were built to benefit, and may need tweaking to target the *meshlet* organization.

Scenario	Traditional	Mesh (1)	Mesh (2)	Mesh (3)	Mesh (4)
Original	349,85	382,20	359,56	389,47	395,76
Orig. & Hoppe	355,01	366,46	354,96	359,20	363,53
Orig. & Forsyth	366,68	370,92	366,25	364,38	367,11
Randomized	49,74	89,60	88,73	89,53	88,51
Rand. & Hoppe	270,42	262,72	261,04	263,19	262,88
Rand. & Forsyth	360,70	363,85	359,45	359,35	364,27

Table 2: Recorded performances (FPS). Best performance per column in bold.

- (1) max_vertices = 32 & max_primitives = 64
- (2) max_vertices = 32 & max_primitives = 128
- (3) max_vertices = 64 & max_primitives = 64
- (4) max_vertices = 64 & max_primitives = 128

This difference in behaviour prompted the writing of a script to allow a statistic analysis of the *meshlets* being created for each scenario and configuration. This scripts reads the buffer files created and retrieves statistics on the primitives of each *meshlet* and their overall vertex locality ratio.

Table 3 and Table 4 present respectively, two of the retrieved statistics by scenario and configuration: the total number of *meshlets* (least amount in bold) and the average miss ratio of each *meshlet* (lowest ratio in bold). The total number of *meshlets* allows us to know how many Mesh shader work groups need to be launched to render the model and the miss ratio - calculated for each *meshlet* by dividing the number of unique indices by the total number of indices - allows us to know how much vertex re-utilization there is on average in each *meshlet*.

By evaluating the the number of *meshlets* in Table 3 and miss ratios in Table 4, the recorded performances in Table 2 start to make more sense. When comparing only the original model and its Hoppe optimized version, one can see that the number of *meshlets* increases and the vertex miss ratio is actually worse (*i.e.*, it's higher). This implies worse performance in the Hoppe optimized version and that can be observed.

However, when checking Forsyth optimization of the original model, the number of *meshlets* is fewer in the optimized and the miss ratio is better, but the performances are worse in all but one configuration (*i.e.*, configuration (2)).

Scenario	Mesh (1)	Mesh (2)	Mesh (3)	Mesh (4)
Original	424006	424006	228417	196424
Orig. & Hoppe	432018	432018	233665	201683
Orig. & Forsyth	423718	423718	226816	195194
Randomized	1269981	1269981	604791	604791
Rand. & Hoppe	505178	505178	264213	232909
Rand. & Forsyth	428282	428282	231388	197922

Table 3: Number of *meshlets*. Lowest amount of *meshlets* for each configuration shown in bold.

Scenario	Mesh (1)	Mesh (2)	Mesh (3)	Mesh (4)
Original	0,498	0,498	0,482	0,481
Orig. & Hoppe	0,499	0,499	0,484	0,486
Orig. & Forsyth	0,482	0,482	0,468	0,467
Randomized	0,996	0,996	0,990	0,990
Rand. & Hoppe	0,584	0,584	0,572	0,574
Rand. & Forsyth	0,486	0,486	0,470	0,472

Table 4: Miss Ratios. Lowest miss ratio for each configuration shown in bold.

As mentioned above, it's not known how the original model may be organized. It's clear that the original model is well organized as the optimizations barely made a difference in the miss ratios recorded. Thus, one can look at the randomized model for a clearer picture.

With the randomized model and its optimizations, the differences in number of *meshlets* and miss ratios is way more significant and the recorded performances also demonstrate said differences. With the randomized model and its optimizations, one can see the impact that these vertex locality optimizers can have on the performance, if the original model isn't very well optimized. In fact, the randomized model and its optimizations demonstrate that the original model of the Powerplant is an outlier and without it a clear trend can be seen.

Figure 32 shows the two graphs demonstrating said trend. When the model is optimized so that the average vertex miss ratio is minimized, every Mesh configuration benefits because it needs fewer *meshlets* to describe a 3D model and, consequently, fewer Mesh shader work groups to render it.

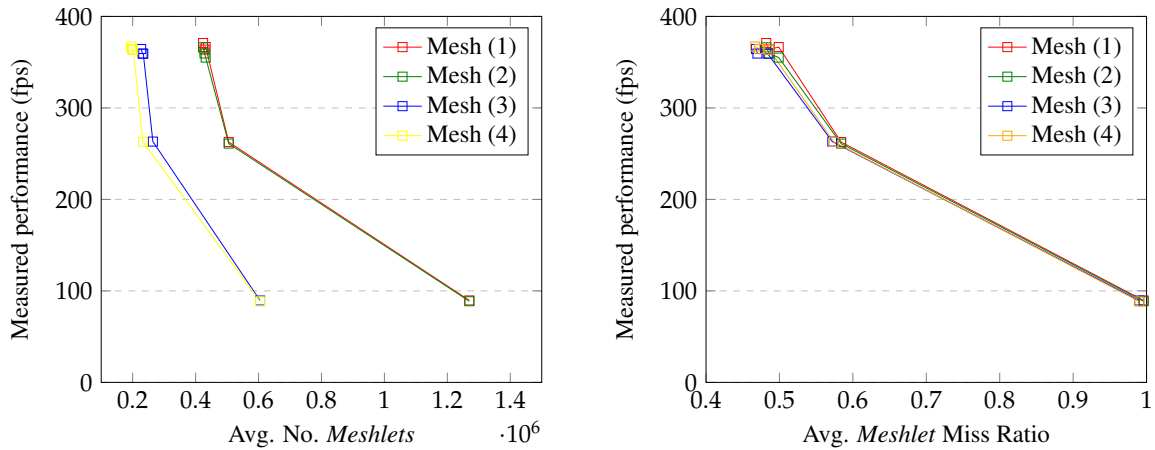


Figure 32: Powerplant No. Meshlets / Miss Ratio to Performance Correlation.

Thus, the vertex locality optimizers are a good tool to use regardless of the type of pipeline, as the principle behind them also benefits the Mesh pipeline. Nonetheless, the algorithms tested seemed to favour more the pipeline they were built to improve, and perhaps different algorithms may prove more suited to *meshlet* structures.

4.4 IMPACT OF WASTING MESH WORK GROUPS

When working with Mesh shaders, the developer must launch a certain number of Mesh shader work groups to complete a specific task. As each Mesh shader work group has limitations in both the number of vertices and the number of primitives it can produce, if a task is large enough it needs to be distributed across several work groups.

In [Subsection 3.3.1](#), a Mesh shader was designed to be able to distribute the work of drawing a grid across several work groups. There's a minimum number of work groups needed to draw each certain sized grid, but after said minimum the shader can dynamically distribute the work through the work groups (*i.e.*, increasing the number of work groups reduces the work done by each one). Thus, one can use said shader to measure the impact that launching more thread groups than necessary has on performance.

[Figure 33](#) shows a graph drawn from such a test, where the minimum work groups necessary were 10000 and that number was increased up to 40000. One can see that initially the impact is very severe, where a few extra work groups can affect the performance significantly. This impact slowly decreases up to half the initial performance with 4 times more work groups. This graph serves to prove that it's in the best interest of the developer the launch of as few work groups as possible, so that each work group can do the most it can for the limits it has.

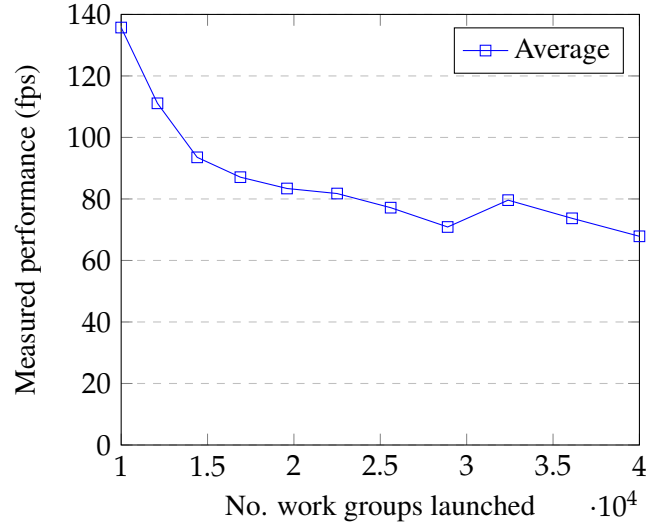


Figure 33: Performance impact of launching more Mesh shader work groups than necessary (*i.e.*, 10000).

With this knowledge of the impact of launching more work groups than necessary, in [Subsection 3.3.1](#) several approaches for distributing the work of drawing a grid by work groups were discussed, in order to find the one that allows the launch of as few work groups as possible.

From the approaches studied, the squared quadrilateral was the first to be implemented. This approach, despite being the theoretical best, can be improved for specific grid configurations where a differently sized quadrilateral may fit better and generate fewer work groups. However, this improvement comes at the cost of a more expensive algorithm in the Task shader, which figures the best size for the quadrilateral through brute force.

As such, to figure out if the benefits in number of work groups of the re-sizable quadrilateral approach outweigh the higher performance cost in the Task shader, two types of tests were executed: a best and worst-case scenario. For the squared approach, the best scenario is when the best size found for the quadrilateral with the re-sizable approach is the square, effectively meaning that the re-sizable approach used more computation to end up with the same solution as the squared approach. The worst case scenario is the opposite, when the re-sizable approach finds a grid size that results in fewer work groups being launched.

[Figure 34](#) represents in a bar graph the four types of tessellation tested with both approaches, with two being a best case scenarios for the squared approach (*i.e.*, both 1500x1500 and 3000x3000 are best divided in 15x15 grids, with 10000 and 40000 being the optimal number of work groups to launch for each, respectively) and two being a best case scenario for the re-sizable approach (*i.e.*, both 1400x1600 and 2800x3200 are best divided with 14x16 grids and not 15x15, with 10000 and 40000 being the optimal number of work groups to launch, respectively). Despite the theoretical presence of best case scenarios for the re-sizable approach, the squared approach edged out the re-sizable approach in every test. These tests were executed several times in alternating order and the results consistently showed that the squared approach achieved better average (shown in the graph) and median performance.

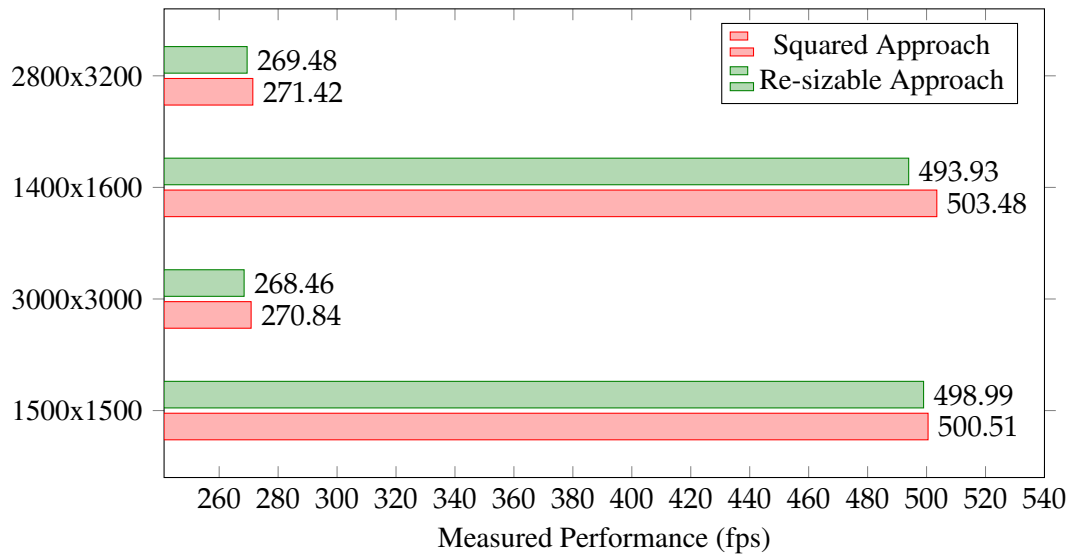


Figure 34: Squared Approach vs Re-sizable (no per-vertex workload added).

Nonetheless, these weren't the only tests taken. To tilt the scales in favour of the re-sizable approach, heavy per-vertex computation was added to the Mesh shader, in order to better understand if in said cases the re-sizable approach, which launches always fewer or the same number of Mesh shader work groups, can perform better than the squared approach. Figure 35 shows the results of said tests. The re-sizable approach comes heavily favoured in these tests, because in the cases where it makes a difference to the number of work groups launched (*i.e.*, both 1400x1600 and 2800x3200 scenarios) it performs about 10% better and in the rest (*i.e.*, scenarios 3000x3000 and 1500x1500), the performance cost is negligible (below 1%).

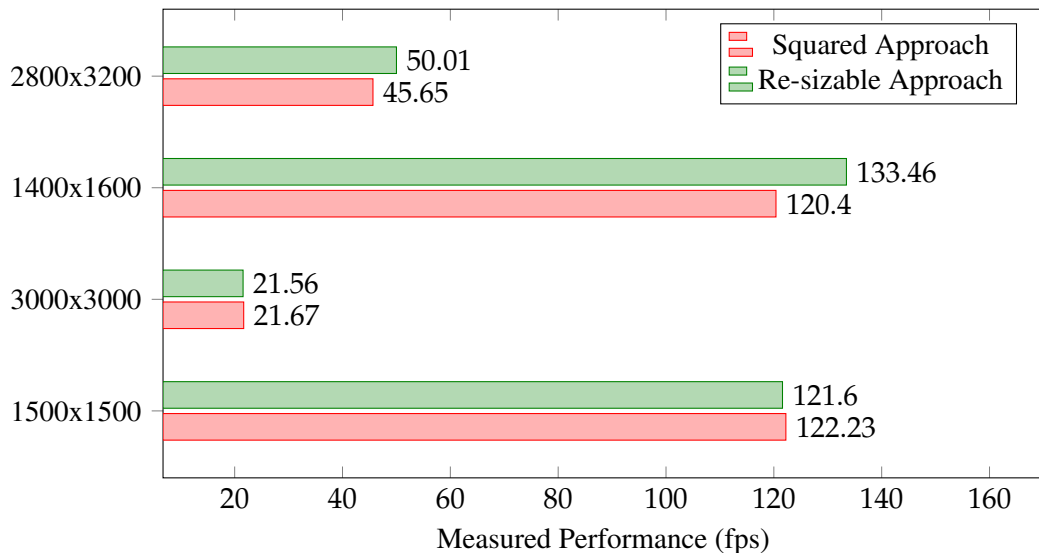


Figure 35: Squared Approach vs Re-sizable (with per-vertex workload added).

With these comparisons, one can see that comparing both approaches is very dependent on the task at hand. If the task at hand doesn't require heavy computation to be done to the generated vertices, the squared approach generates a fairly good and inexpensive approximation to the optimal number of work groups to launch. If the

task requires per-vertex operations that may significantly weight on the overall performance, than the slowest re-sizable grid approach may provide a lower number of work groups to launch than the squared approach, which may result in better performance.

4.5 IMPACT OF THE LOCAL_SIZE PARAMETER

As mentioned in [Subsection 2.3.3](#), when writing a Mesh shader the developer must choose the number of threads each work group will work with. When doing so, the developer is limited to a maximum (*i.e.*, `MAX_MESH_WORK_GROUP_SIZE_NV`) which for the testing hardware is 32. In this section, the impact that this parameter has on the final performance of the Mesh shaders is explored.

The number of threads to choose should not be any number. The maximum number of threads being 32 is not by chance. In NVIDIA's terms, each [Streaming Multiprocessor \(SM\)](#) has units called *Warps*, with each *Warp* being able to run 32 threads at a time. One may assume that when this number of threads is chosen in the Mesh shader, each work group is assigned to a *Warp*.

It's still unclear though what happens when the number is not set to the maximum. Can a *Warp* handle more than one work group at a time, if said work group requires a smaller number of threads? If it does, the number of work groups chosen should be a divisor of 32, to ensure that no processing power is lost.

Nonetheless, in the end what matters is the performance obtained by using the different number of threads. To test the performance obtained with each number of threads, other parameters that may influence the result must be nullified. To test this, different models were rendered for three sizes of thread groups (*i.e.*, groups of 8, 16 and 32 threads), with each differently sized group also being tested with several different maximum number of vertices that each can handle, so that different amounts of work can be tested. The maximum number of primitives was set to the maximum possible (*i.e.*, 512), as the actual number of primitives emitted is dependent on the vertex re-utilization of each *meshlet*.

The first model tested was the famous Stanford Dragon model, composed of 438929 vertices and 871306 primitives. This model contains no materials, thus needing a single draw call to be rendered by our code. [Table 5](#) displays the [FPS](#) recorded with each number of threads for said model. For each amount of work (*i.e.*, maximum number of vertices to emit), in bold is the number of threads that registered the higher [FPS](#).

One can see that for a number of vertices greater or equal to 32, the maximum group size of 32 always registers better performance. Although, when the number of vertices is inferior to 32, the local size of 16 registered better performance both for 16 vertices, as well as 8 vertices.

<i>Dragon</i>	MV = 8	MV = 16	MV = 32	MV = 64	MV = 128	MV = 256
Local Size = 8	388	639	788	705	576	361
Local Size = 16	391	693	899	967	828	584
Local Size = 32	387	687	928	1083	1062	832

(MV) Maximum Vertices Emitted by Each Work Group

Table 5: Recorded Performance with each Local Size (Stanford Dragon model).

Nonetheless, the best performance recorded for each scenario doesn't tell the whole story. One should look as well to the differences in performance recorded. Figure 36 translates Table 5 into a bar graph, where the performance differences in each scenario can be better visualized.

Clearly, the potential performance that can be gained by using the size of the *Warps* (i.e., 32 threads) as local size is significantly higher than the performance that may be lost when the task is too trivial.

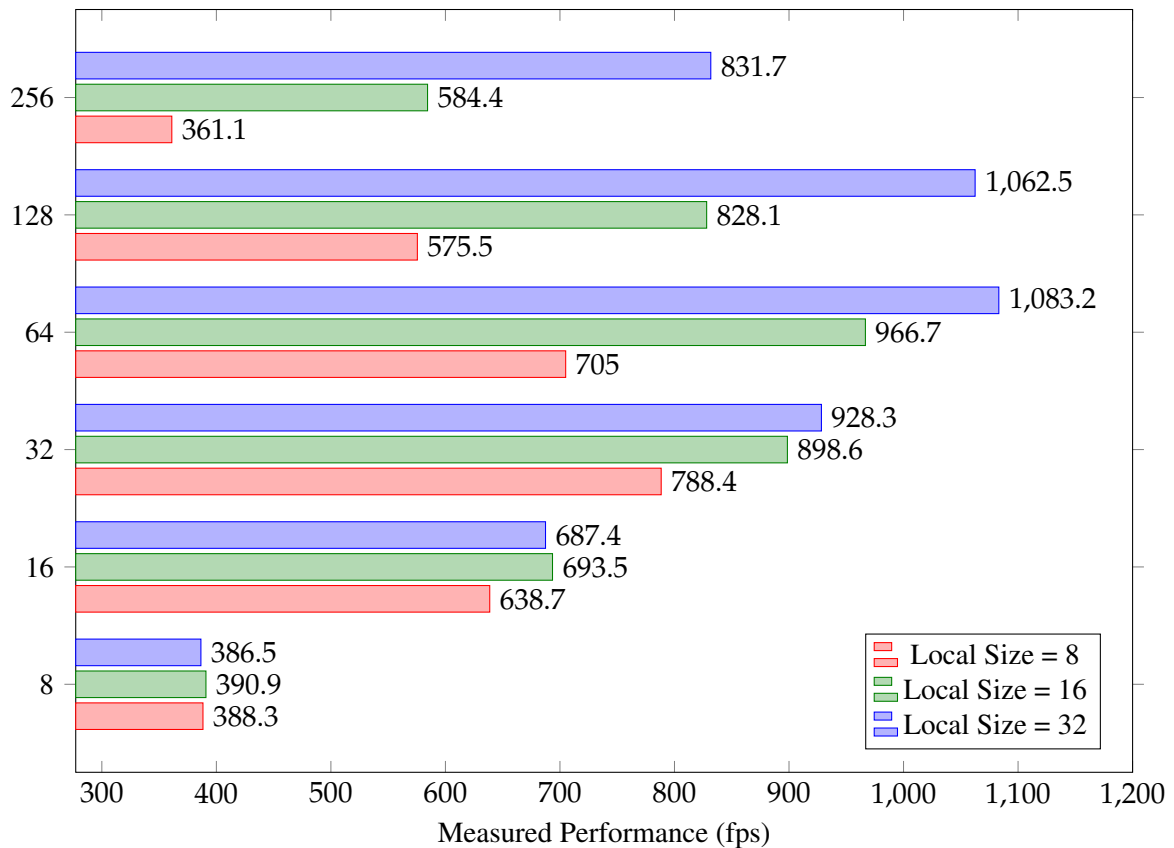


Figure 36: Recorded Performance with each Local Size (Stanford Dragon model).

This behaviour is also observable with other models. In addition to the Stanford Dragon, both the Powerplant (of 10614919 vertices and 12759246 triangles) and the San Miguel (of 5933231 and 9963191 triangles) models were tested. Both these models have several materials, requiring several draw calls to be rendered. Additionally, the San Miguel has textures, which adds to the information passed from the Mesh shader to the Fragment shader.

In Table 6 and Table 7, one can see the measured performances for the Powerplant and San Miguel models, respectively. These share the trend observed with the Stanford Dragon model, with the only notable difference in the shader with 8 maximum vertices. With 8 maximum vertices, in these two models the better performing amount of threads was 8, instead of the 16 suggested by the Stanford Dragon model.

As mentioned before, one must look at the differences in performance in each case. In every model, the performance obtained for 8 maximum vertices by each thread number is almost the same, with the difference being so small that it could be attributed to margin of error.

In theory, though, for 8 vertices both the groups of 16 and 32 threads should perform worse than the 8 threaded group, because every shader whose identification doesn't correlate directly to a vertex gets assigned the last possible vertex (*i.e.*, every thread whose index is 8 or above is charged of emitting the 8th vertex). Thus, every thread ends up doing the same amount of work and the unnecessary *threads* end up emitting the same vertex to the same memory address, which may lead to *threads* waiting to write to said address.

<i>Powerplant</i>	MV = 8	MV = 16	MV = 32	MV = 64	MV = 128	MV = 256
Local Size = 8	76	142	205	187	137	76
Local Size = 16	75	161	242	284	224	132
Local Size = 32	74	157	264	339	330	217

Table 6: Recorded Performance with each Local Size (Powerplant model).

<i>San Miguel</i>	MV = 8	MV = 16	MV = 32	MV = 64	MV = 128	MV = 256
Local Size = 8	74	134	190	163	123	70
Local Size = 16	73	151	240	253	205	125
Local Size = 32	73	149	263	336	309	205

Table 7: Recorded Performance with each Local Size (San Miguel model).

With these performance metrics for two additional 3D models, one can plot performance graphs to try and observe the tendencies described. As can be seen in Figure 37, every model ends up with a similar graph. These graphs show that once the maximum vertices get to 32, the recommended number of threads in the work group is 32.

In essence, once the work can be distributed without having threads doing unnecessary extra work, the maximum amount of threads of a *Warp* should be chosen for the `local_size`. Because the graphics API didn't allow for a `local_size` greater than the size of a *Warp*, these values couldn't be tested.

Thus, for now, when a developer is writing a Mesh shader, unless the amount of work to be done by work group is minuscule, the `local_size` parameter should be set to the number of *threads* run by a *Warp*: 32.

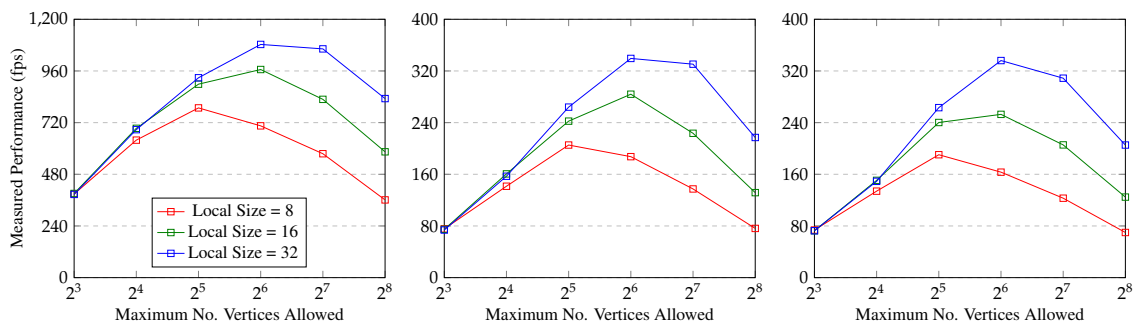


Figure 37: Performance impact of using different values for the `local_size` parameter in the Stanford Dragon, Powerplant and San Miguel models, respectively.

4.6 IMPACT OF THE MAX_VERTICES PARAMETER

With the impact of the `local_size` parameter studied in the previous section, in this section the impact of the `max_vertices` parameter is studied.

The `max_vertices` parameter sets the amount of vertices each Mesh work group emits. To test this parameter, one can measure the performance obtained in different models for different values. The different values for the `max_vertices` are different multiples of 32, to ensure that every thread in the work group emits a unique set of vertices.

In this section, the tests were carried out with four different 3D models: Stanford Dragon, Powerplant, Rungholt and Lucy. As for the other parameters that might influence the measured performance, the `local_size` was kept at 32 and the `max_primitives` was set to one of these values: 40, 84, 126, 254 and 510. The values for the `max_primitives` weren't chosen at random but based on memory alignment, which is explored in [Section 4.7](#).

Starting with the Lucy 3D model (composed by 14027872 vertices and 28055742 triangles), [Table 8](#) shows FPS recorded for the different `max_vertices` in different scenarios. For every scenario (*i.e.*, for each different maximum number of primitives allowed) the configuration with `max_vertices` set to 64 achieved the best result. Additionally, until the maximum number of 64 the recorded performance continuously increases and after said number it continuously decreases.

<i>Lucy</i>	MP = 40	MP = 84	MP = 126	MP = 254	MP = 510
MV = 8	28,1	27,2	26,3	22,5	14,5
MV = 16	58,6	57,3	54,4	46,5	31,0
MV = 32	85,9	81,5	78,6	68,0	52,7
MV = 64	90,3	88,8	86,7	80,8	67,7
MV = 128	68,6	70,9	70,5	70,2	63,7
MV = 256	36,8	40,3	40,9	41,0	40,4

Table 8: Recorded performance (fps) for different `max_vertices` values (Lucy model).

(MV) `max_vertices`

(MP) `max_primitives`

This results can be best seen in [Figure 38](#), where for every scenario the same pattern can be observed. Besides the 64 vertices achieving the best performance in every scenario, the 8 maximum vertices recorded the worst. The only thing that changes in the ordering of the scenarios is the second and third best performances exchanging places, as well as the third and fourth. The explanation to this phenomenon is explored in [Section 4.7](#).

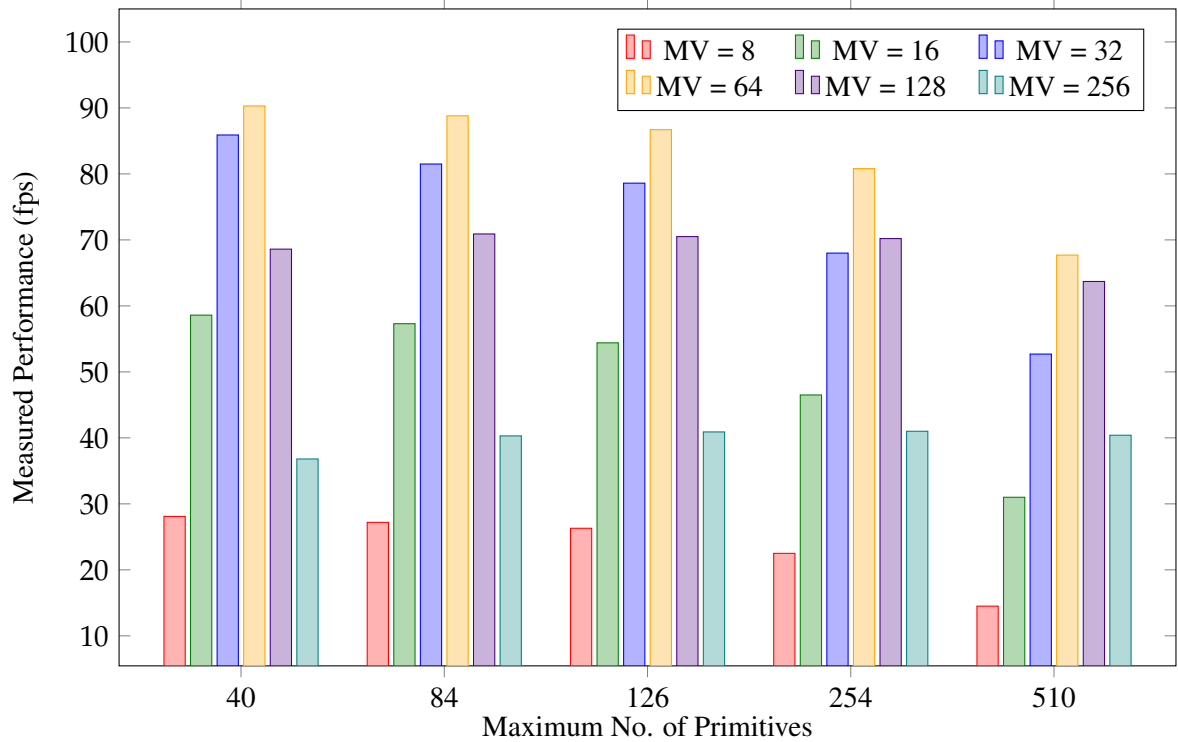


Figure 38: Recorded Performance by Max Vertices value (Lucy model).

Continuing to the Stanford Dragon and Powerplant 3D models, the hegemony of `max_vertices=64` ends. Whilst having a maximum of 64 vertices still results in good performance across the board, it no longer edges the others in every single scenario. With the Stanford Dragon model (graphed in Figure 39), for a maximum number of primitives of 254, using 128 vertices resulted in better performance, although by not a big margin. With the Powerplant model (graphed in Figure 40), however, for `max_primitives=40`, the Mesh shader with `max_vertices=32` achieves the best results by a considerable margin.

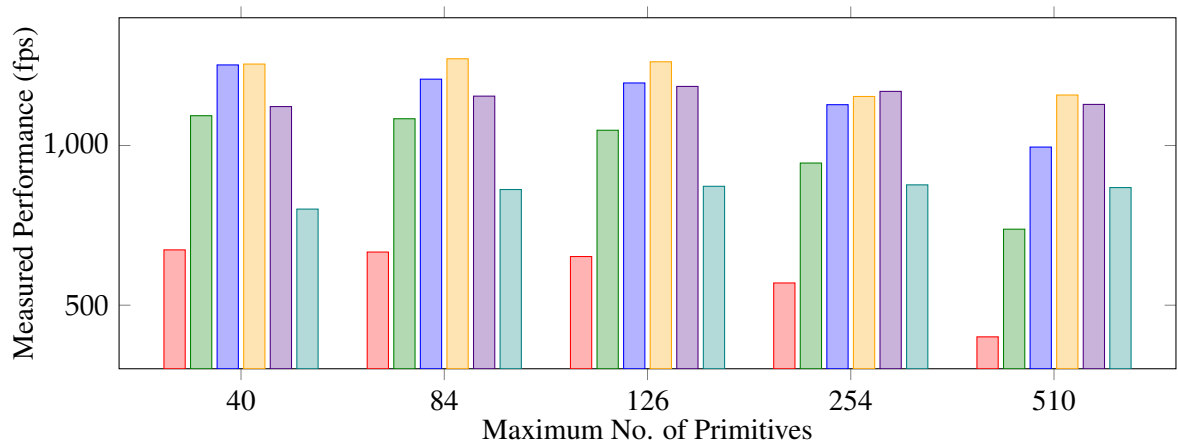


Figure 39: Recorded Performance by Max Vertices value (Stanford Dragon model).

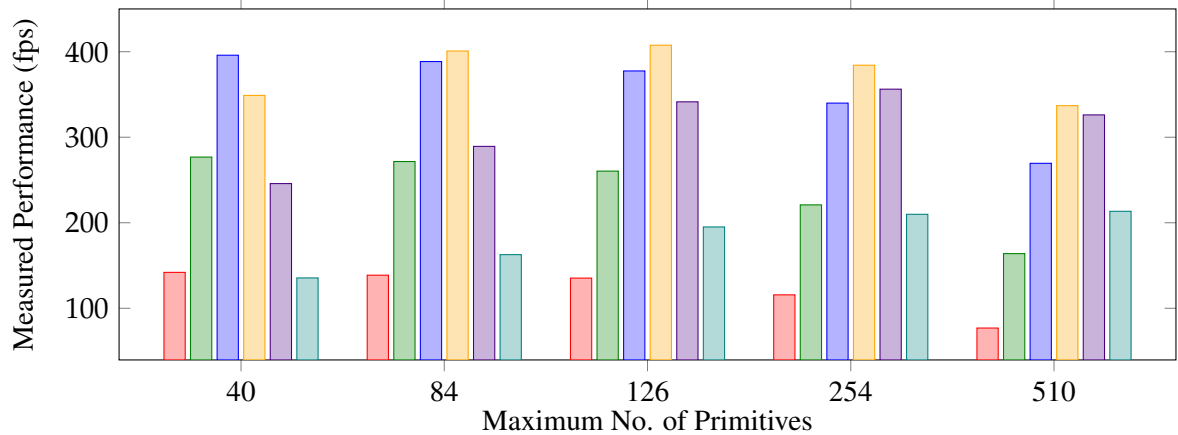


Figure 40: Recorded Performance by Max Vertices value (Powerplant model).

What remains the same is the inverted parabola behaviour that can be noticed when observing each maximum number of vertices across different scenarios. By inverting the bar graph of [Figure 40](#), one gets the very telling graph presented in [Figure 41](#).

One can observe that for each maximum number of vertices, the bars form a curve akin to an upside down parabola. For the configurations with 32 or fewer vertices, the right side of the parabola can be seen with the performance decreasing with increasing intensity. For the configurations with 64 or 128 vertices, both sides of the parabola can be seen. For the configurations with 256 vertices, only the left side of the parabola can be seen, which the performance increasing less and less with each step.

This indicates that for every `max_vertices` value there is one `max_primitives` that best complements it. When analysing the *meshlets* created for this test with the Powerplant model, in almost every configuration, increasing the maximum number of primitives did not result in more primitives being rendered by each work group. This means that when creating the *meshlets*, the algorithm is usually stopping because it hit the maximum number of vertices rather than the maximum amount of primitives.

This signifies that the `max_vertices` parameter has a bigger impact on the creation of the *meshlets* than the `max_primitives` parameter. One can see this as regardless of the maximum number of primitives, the configurations with 8 maximum vertices produce mediocre performance numbers, with the configuration with 256 maximum vertices not fairing much better.

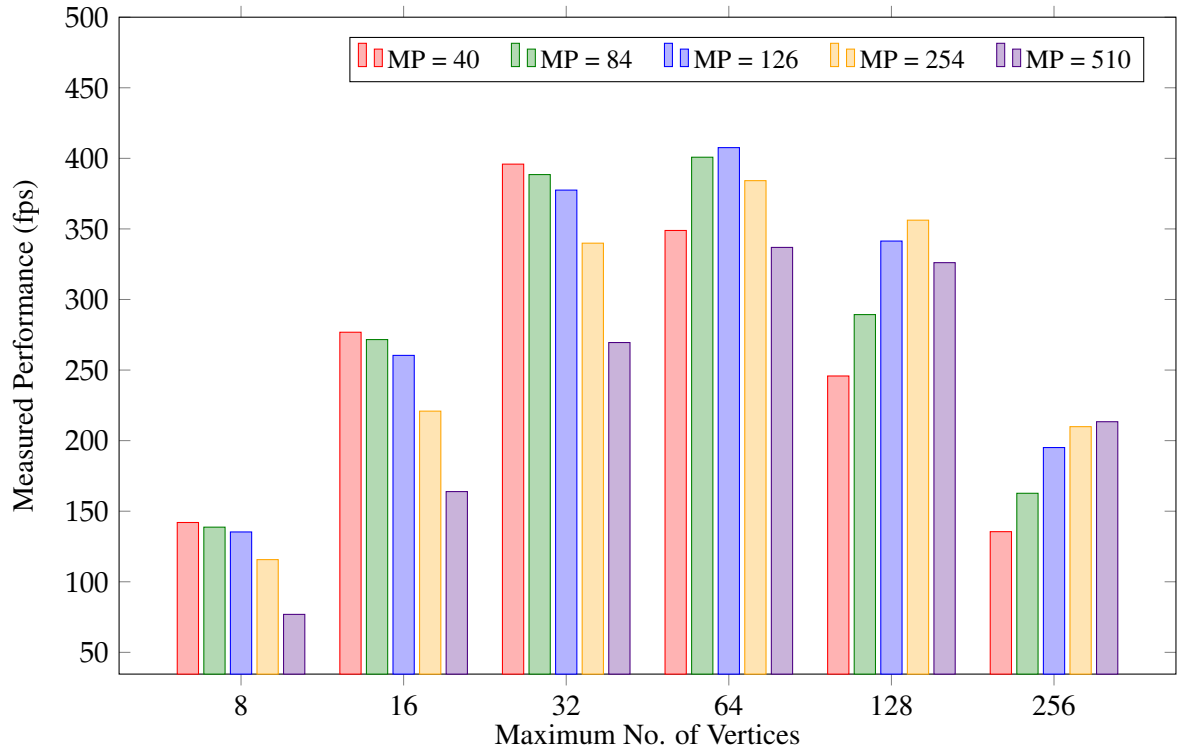


Figure 41: Recorded Performance by Max Primitives value (Powerplant model).

Nonetheless, when considering only the highest recorded performance, in all three models the value set for the maximum amount of vertices was 64. This only changes in our fourth evaluated 3D model: the Rungholt (composed of 12308528 vertices and 6704264 triangles).

In Figure 42, one can see that the best performant across all scenarios was the configuration of 32 vertices, in the scenario where the maximum number of primitives was set to 40. However, with 64 vertices the performance is almost indistinguishable, except for 254 and 510 maximum primitives.

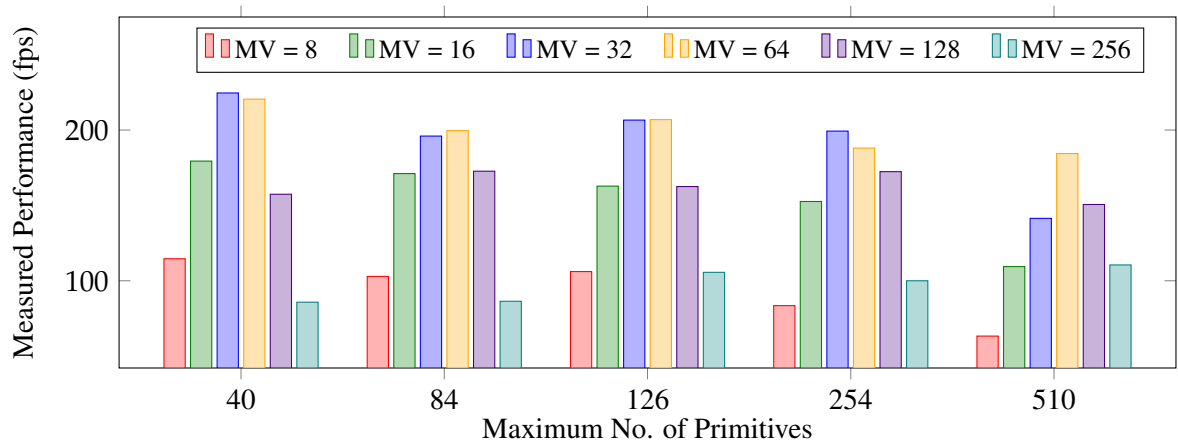


Figure 42: Recorded Performance by Max Vertices value (Rungholt model).

In the end, in most cases it seems that a value of 64 maximum vertices will produce the best results. However, 32 maximum vertices was able to be the best for the Rungholt model and may be for some others, so it should also be taken in consideration.

4.7 IMPACT OF THE MAX_PRIMITIVES PARAMETER

With both the `local_size` and `max_vertices` parameters chosen, the last parameter `max_primitives` must also be studied. This parameter is similar to the `max_primitives` and is directly linked to it. As ascertained in [Section 4.6](#), for each `max_vertices` parameters there's an optimal `max_primitives` value that results in the best performance.

As such, for each model studied in this section, three `max_vertices` values were tested with several `max_primitives`. In this section, three differently constructed 3D models were studied: Lucy, Powerplant and Rungholt.

As for the values chosen for the `max_primitives`, initially the values chosen were those recommended by NVIDIA ([Mours, 2018](#)) : 40, 84, 126, 254 and 510. These values are favoured due to memory allocation, which is done in 128 byte blocks, with a 4 byte fixed header. In order to maximize the use of these blocks, the amount of primitives to use can be calculated by:

1. 1 block = 128 bytes -> (128 bytes - 4 byte header) / 3 indices by primitive = 41 primitives
2. 2 blocks = 256 bytes -> (256 bytes - 4 byte header) / 3 indices by primitive = 84 primitives
3. 3 blocks = 384 bytes -> (384 bytes - 4 byte header) / 3 indices by primitive = 126 primitives
4. 6 blocks = 768 bytes -> (768 bytes - 4 byte header) / 3 indices by primitive = 254 primitives
5. 12 blocks = 1536 bytes -> (1536 bytes - 4 byte header) / 3 indices by primitive = 510 primitives

From these calculations, one can see that 41 primitives can fit in a single 128 byte block, and not only 40, as recommended by NVIDIA. Additionally, there are values for the blocks to use besides those suggested (e.g., 4, 5, 7, etc.), which enable a more granular choice for the `max_primitives` value (e.g., 169, 212, 297, etc.).

Thus, to verify NVIDIA's suggestions and to measure the impact that memory alignment has on the performance metrics, between the values suggested by NVIDIA four equidistant values were chosen and measured as well.

Starting with the Lucy model, [Figure 43](#) represents in a bar graph the measured FPS in the different scenarios. The colors chosen for the graph bars represent how many blocks each configuration requires (e.g., every red bar requires a single 128 byte block, whilst the green ones require two). The opacity of each bar represents how much of the capacity of the blocks is being used by the configuration (e.g., the configuration with 10 maximum vertices uses a full 128 byte block that could handle up to 41 primitives, so its color is faint, whilst the 40 vertices configuration is brightly colored).

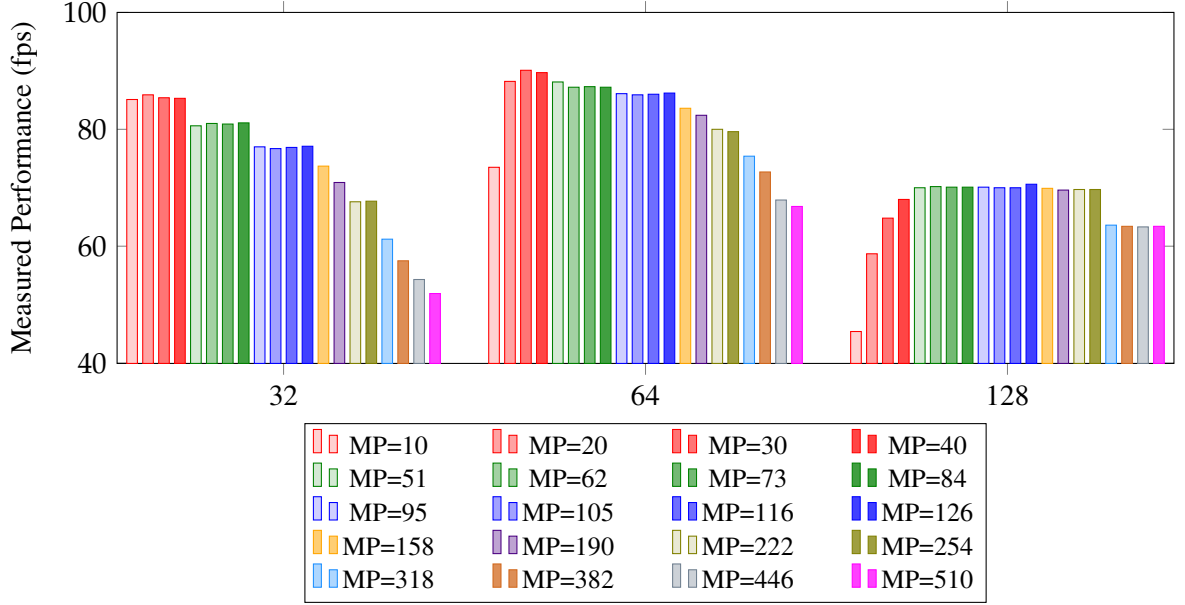


Figure 43: Recorded Performance by Max Primitives value (Lucy model).

With 32 `max_vertices` scenario, the best performances were recorded for the configurations colored in red (*i.e.*, those with values for the maximum vertices below 41). One can also see that in this scenario, the configurations between 51 and 84 performed very similarly, as well as the configurations between 95 and 126. In fact, with every change in color, the recorded performance decreases, which implies that each additional allocated block is affecting the final performance.

However, for the remainder scenarios presented (*i.e.*, when `max_vertices` is set to 64 and 128), one can see significant performance differences amongst the red configurations, which allocate the same amount of memory. This is due to, inadvertently, the `max_vertices`=32 scenario being ideal to measure the impact of memory alignment, as every configuration does the same amount of work but allocates different amounts of memory.

To better explain this behaviour, the sizes of the *meshlets* created for each maximum amount of vertices were measured, for every model, and are presented in Table 9.

	Lucy			Rungholt			Powerplant		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
MV = 32	6	10	12	2	15	16	2	24	48
MV = 64	5	21	22	2	29	32	2	50	94
MV = 128	2	42	44	2	54	64	4	100	192

Table 9: *Meshlet* statistics for different models in the different scenarios.

(MV) `max_vertices`

One can see that the Lucy model, for a maximum amount of vertices of 32, enables only a maximum amount of primitives of 12 per *meshlet*. As such, in this scenario, only the configuration with `max_primitives`=10 is emitting a slightly different amount of primitives per *meshlet* (*i.e.*, this configuration emits 10 vertices, whilst all others emit 12, regardless of the chosen `max_primitives` value). Thus, this scenario is perfect to show

that using a higher value for the `max_primitives` parameter than the actual number of primitives emitted can affect the final performance, if it leads to the allocation of unnecessary memory blocks.

For the 64 and 128 maximum vertices scenarios, the maximum amount of primitives the Lucy model enables is 22 and 44, respectively. With this increase of performance, one can see that no longer does every configuration in red perform about the same. In fact, in the 128 scenario, the green configurations perform better than any red one because the advantage of emitting 44 vertices outweighs the drawback of allocating one more block of memory.

Advancing to the Rungholt model, Figure 44 paints the same picture as seen with the Lucy model. Even though the Rungholt model allows for more vertex reuse than the Lucy model (*i.e.*, the Lucy shares no vertex indices, with every triangle being unique, while the Rungholt model shares a couple of indices every two primitives, as it's built with unique quads in mind), the results share the behaviour already seen.

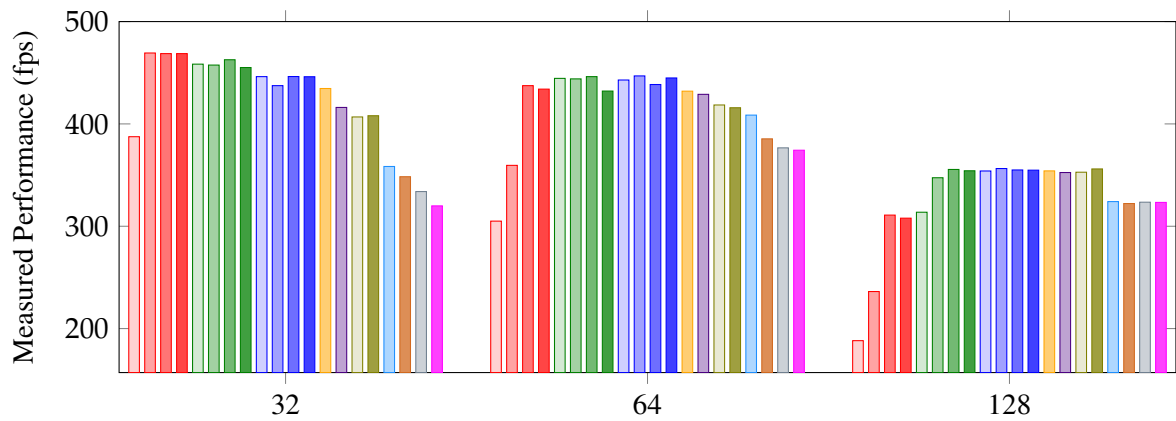


Figure 44: Recorded Performance by Max Primitives value (Rungholt model).

The Rungholt model allows for a maximum number of primitives of 16, 32 and 64 for a maximum amount of vertices of 32, 64 and 128, respectively. In the graph, one can see that until said value is covered by each work group the measured performance is not optimal. On top of that, once said value is covered, allocating more memory blocks than necessary harms the measured performance, even though said behaviour becomes more forgiving with a higher number of maximum vertices.

All of these behaviours were prevalent in the Lucy model and are present too with the Rungholt model as well. The Rungholt model shows a bit more variance across configurations with the same amount of emitted vertices and allocated blocks of memory, but said variations fall under the margin of error.

Finally, the Powerplant model, graphed in Figure 45, presents an even clearer picture of these behaviours. Of all the models seen in this section, the Powerplant is the one which allows for more vertex reuse by a lot. With the Powerplant model, for a maximum vertex count of 32, 64 and 128, the maximum number of primitives a work group can emit is 48, 94 and 192, respectively.

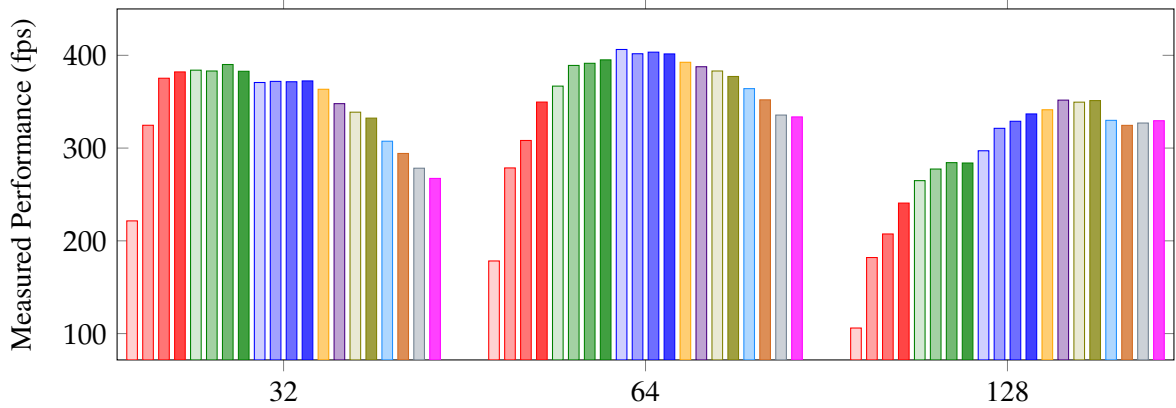


Figure 45: Recorded Performance by Max Primitives value (Powerplant model).

However, unlike what is observed with the Lucy and Rungholt models, the maximum amount of primitives emitted is way disparate from the average amount of primitives emitted by each work group. For the Powerplant model, the configurations with the highest amount of vertices per *meshlet* recorded an average amount of primitives of 24, 54 and 100, for the scenarios with a maximum amount of vertices of 32, 64 and 128, respectively. As big of a disparity was not seen with the Lucy and Rungholt models, whose average primitives emitted by work group can be seen in [Table 9](#).

Because of this disparity, increasing the `max_primitives` parameter up to the maximum number of primitives possible doesn't result in a linear increase in performance, as the average amount of primitives emitted doesn't increase that much. However, it still results in performance gains as the number of *meshlets* can still be reduced.

In the end, when setting the `max_primitives` the goal is to allocate just enough memory blocks to be able to emit the least amount of *meshlets* possible. Thus, one strategy to get the optimal value for this parameter is to create the *meshlets* using only the `max_vertices` parameter as the limiting factor to the size of the *meshlets*, calculate the maximum primitives emitted in that scenario and set the `max_primitives` to said value.

4.8 RENDERING 3D MODELS WITH THE MESH SHADER PIPELINE

With a better understanding of the best values for the Mesh shader parameters, a fair performance comparison can now be made between the two different pipelines: the traditional pipeline and the mesh pipeline. In this section, this performance comparison will be made by rendering different 3D models with the two pipelines.

For this comparison, six different models were chosen, each with different characteristics. For each model and pipeline, a few runs were executed to avoid taking conclusions on possibly abnormal results. Additionally, three different mesh pipelines were tested, each with different `max_vertices` values (*i.e.*, 32, 64 and 128 vertices). For each of these `max_vertices` values, the maximum amount of primitives allowed by the model was set as the `max_primitives` value. As for the `local_size` parameter, it was kept at 32.

Starting with the two models which showed more variation in their results, [Figure 46](#) shows the measured performances for the Sponza and Dragon models. These models record over 1000 FPS in every run, so higher margins of error must be considered.

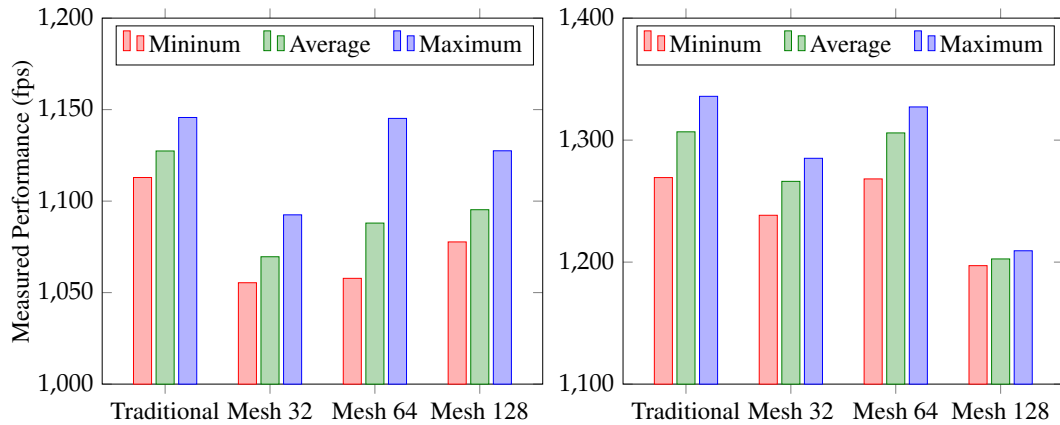


Figure 46: Measured Performance with both pipelines for the Sponza (left) and Dragon (right) models.

Despite the variations in the measured FPS, tendencies can still be observed. It can be observed that the Sponza model favours the traditional pipeline, as the average and minimum performance recorded for this pipeline outperformed every configuration of the mesh pipeline. However, in one of the mesh configurations (*i.e.*, `max_vertices=64`) the maximum performance recorded was able to match the maximum performance of the traditional pipeline. This was an abnormal result, as only a single run was able to obtain such a high performance.

As for the Dragon model, the mesh pipeline with 64 maximum vertices matches the performance of the traditional pipeline. The mesh pipeline recorded a bit worse performance in every stat, but with such high FPS, these differences cannot be considered significant.

But why does the Sponza model perform consistently worse whilst the Dragon model was able to get matching performances? Does the Dragon model allow for more vertex reuse than the Sponza? Actually, this couldn't be further than the truth. The Dragon model, has almost no reuse whatsoever (*i.e.*, the measured miss ratio was 0.999), whilst the Sponza model actually has a miss ratio of 0.416.

Thus, this difference must be explained by another difference of the models. The main difference between these models is that the Sponza model is composed of several materials, whilst the Dragon model has no materials whatsoever. With each material, a different launch command must be executed to deploy a set number of work groups that could be bundled within a single launch command, were it not for the different material properties.

This behaviour is not unique to these two models. Advancing to two other models that share similar characteristics (but have considerably higher number of vertices), the variations within the configurations are reduced but the tendencies remain the same. In Figure 47, it's presented the measured performances for the San Miguel and Lucy models, which can be considered bigger versions of the Sponza and Dragon models, respectively.

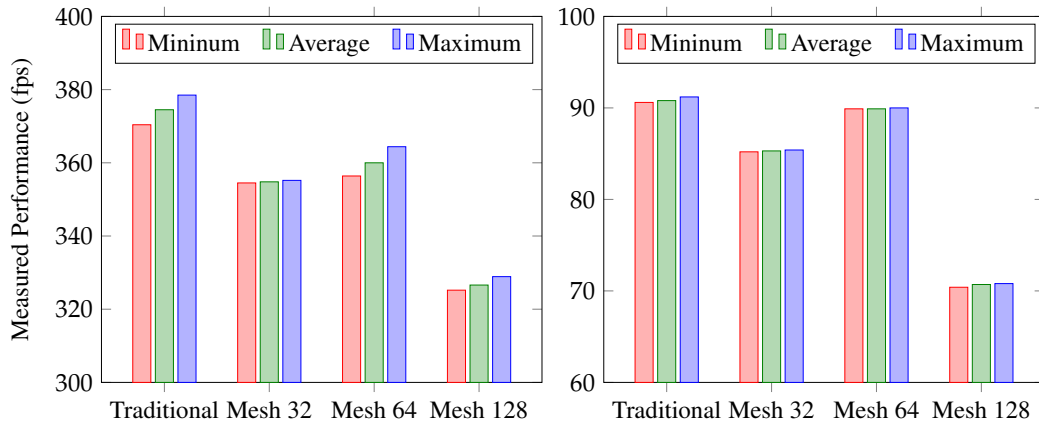


Figure 47: Measured Performance with both pipelines for the San Miguel (left) and Lucy (right) models.

With the San Miguel model, no mesh pipeline could match the performance of the traditional, and with the Lucy model, one configuration of the mesh pipeline was able to almost match the traditional pipeline, with the others falling behind. This was the exact same behaviour observed with the Sponza and Dragon models.

However, these results don't imply that when using materials Mesh shaders should automatically be discarded. This implication can be easily disproved by Figure 48, which shows the results obtained for the Rungholt and Powerplant models. Both these models have several materials and can perform better under a mesh pipeline.

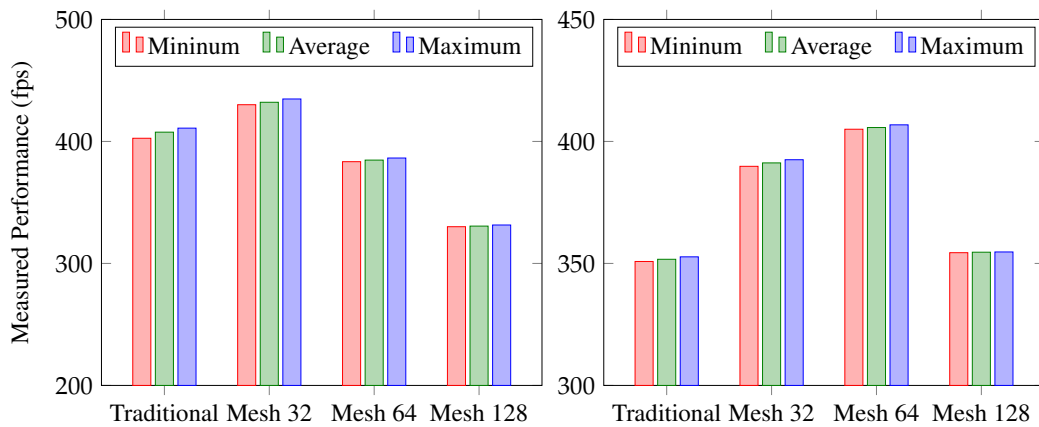


Figure 48: Measured Performance with both pipelines for the Rungholt (left) and Powerplant (right) models.

The Rungholt model performs fairly better under a mesh pipeline with a maximum amount of vertices of 32 than with the traditional pipeline, while being composed of several textured materials, just as the San Miguel and Sponza models. On top of that, the Rungholt has the worse miss ratio of the three at 0.666, compared to the 0.416 and 0.524 of the Sponza and San Miguel models.

As such, the only possible explanation for this difference is the amount of materials and the amount of *meshlets* each emits. In Table 10, the number of materials of each model can be seen, as well as the average number of *meshlets* emitted for each material.

Both the Rungholt and the Powerplant models make use of more materials than the Sponza models and are able to perform better with a mesh pipeline, whilst the Sponza does not. Thus, the value to look for is not the amount of materials but the average number of *meshlets* emitted with each launch call. With each model that has

	Sponza	San Miguel	Rungholt	Powerplant
No. Materials	25	287	84	56
Avg. <i>Meshlets</i> per Material	90.68	682.88	1573.95	3507.57

Table 10: Amount of materials and *meshlets* per material for the different models.

an higher amount of geometry per material, the mesh pipeline performs better when compared to the traditional model. Clearly, the wasting of launch calls with non-complex geometry affects the mesh pipeline performance.

As for the Lucy and Dragon model, which make use of a single launch call, they do not achieve better performance because of the non-existent vertex reuse. Both these models were constructed in the same way and share the same property which deters vertex reuse: split normals. With each vertex of a primitive having their normal equal to the normal of said primitive, then vertex reuse with adjacent primitives can only happen if said primitives share the same normal. As these models contain almost no flat surfaces composed of more than one primitive, then virtually no vertex reuse can happen.

To conclude, the mesh pipeline won't improve the rendering times for all types of 3D models. In fact, the mesh pipeline appears to only benefit from models with high amounts of geometry which share significant amounts of vertices and don't require wasteful launch calls.

4.9 TESSELLATION WITH THE MESH SHADER PIPELINE

With the development made in [Section 3.3](#), the performances of the mesh and traditional pipelines can be compared for the use case of tessellation.

Starting with the tessellation of a single quad, [Figure 49](#) presents the results obtained with both pipelines for three different tessellation levels: 16 (shown on the left), 32 (on the middle) and 64 (on the right). Tessellating a single quad presents no real challenge to the testing hardware, which registers performances above 1000 FPS. With performance metrics in these ranges, there are a lot of variation amongst the results and, consequently, several runs were executed to mitigate them.

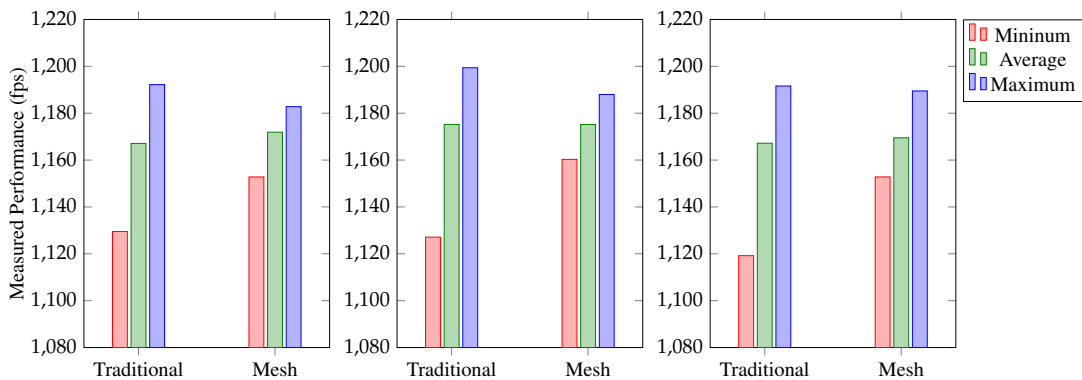


Figure 49: Measured performance of both pipelines when tessellating a quad with different tessellation levels: 16 (left), 32 (middle) and 64 (right).

Despite these measures, the results can't produce differences between the pipelines, except for the consistency. The mesh pipeline presented less variations in its results, as can be seen by the differences between the maximum and minimum registered performances. Besides the variations, this test does not produce significant differences across the pipelines, as on average every pipeline performed about the same in every scenario.

This implies that a fair comparison can't be made of these pipelines for non-demanding workloads. Increasing the workload could be done by increasing the tessellation levels, but this would only work with the mesh pipeline, as the traditional pipeline is limited to 64 levels of tessellation, while the mesh pipeline is only limited by the number of work groups that can be launched (*i.e.*, in the testing hardware is 65535), which for a configuration of 64 `max_vertices` results in a theoretical maximum amount of tessellation of 1785x1785 (*i.e.*, $\text{floor}(\sqrt{65535}) * \text{floor}(\sqrt{64} - 1) = 1785$).

To increase the workload of the traditional pipeline, instances can be used instead. For example, 4 instances of a quad with 64 tessellation levels produces the same amount of primitives as 1 instance of a quad with 128 tessellation levels. With this strategy, Figure 50 shows the results obtained when increasing the levels from 256 (which requires 16 instances of the traditional pipeline) to 1024 (which requires 256 instances).

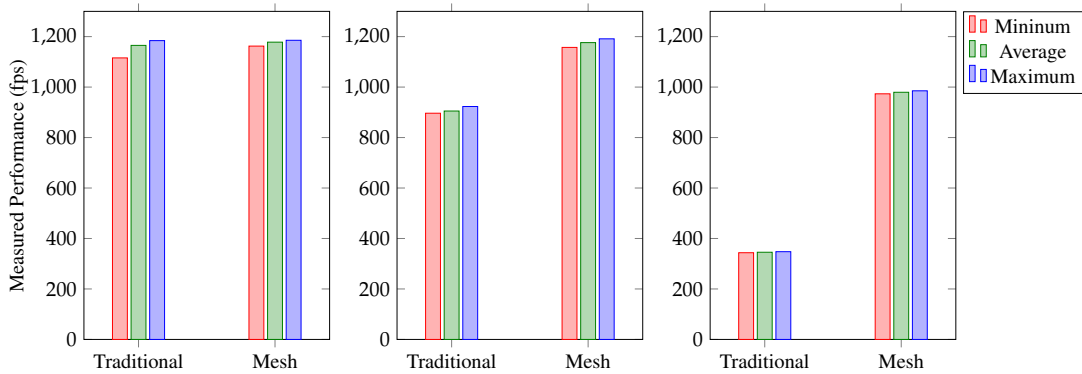


Figure 50: Measured performance of both pipelines when tessellating a quad with different tessellation levels: 256 (left), 512 (middle) and 1024 (right).

With the increased levels, both the decline of the performance with the increase in tessellation can be seen, as well as meaningful differences between the pipelines. For 256 tessellation levels, both pipelines continue performing about the same, but for 512 tessellation levels the mesh pipeline achieves about 250 more FPS than the traditional pipeline and for 1024 tessellation levels about 600 FPS more.

This trend continues when increasing the maximum amount of tessellation above what a single task shader can order (*i.e.*, a single Task shader can emit a maximum of 65535 Mesh shader work groups in the testing hardware). Using the same strategy used with the traditional pipeline's instances, to virtually increase the amount of tessellation, several Task shaders can be launched to achieve the levels of 2048, 4096 and 8192. These results can be seen in Figure 51.

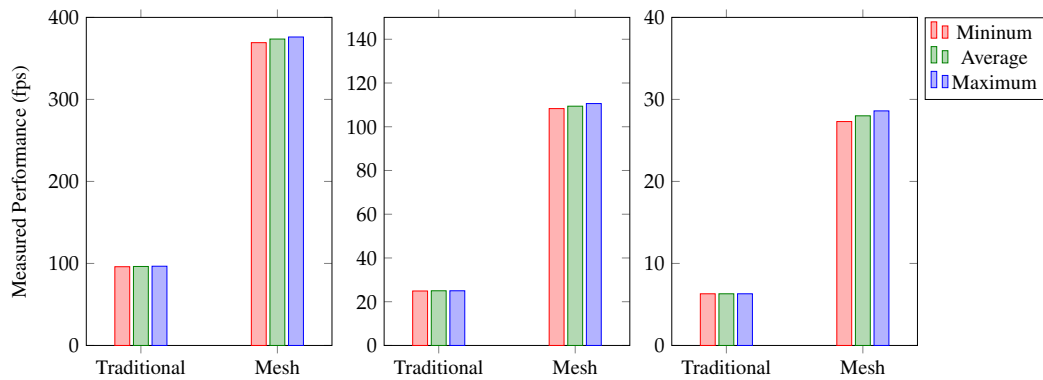


Figure 51: Measured performance of both pipelines when tessellating a quad with different tessellation levels: 2048 (left), 4096 (middle) and 8192 (right).

Despite the use of several launch calls, the mesh pipeline continues to have an advantage over the traditional pipeline. This performance advantage can be attributed to having the mesh pipeline tessellating up to 1024 tessellation levels, instead of the limited 64 of the tessellation shader. When the mesh shader is artificially limited to 64 tessellation levels and the amount of launch calls matched to the number of instances of the traditional pipeline, the results are entirely different (as pictured in Figure 52).

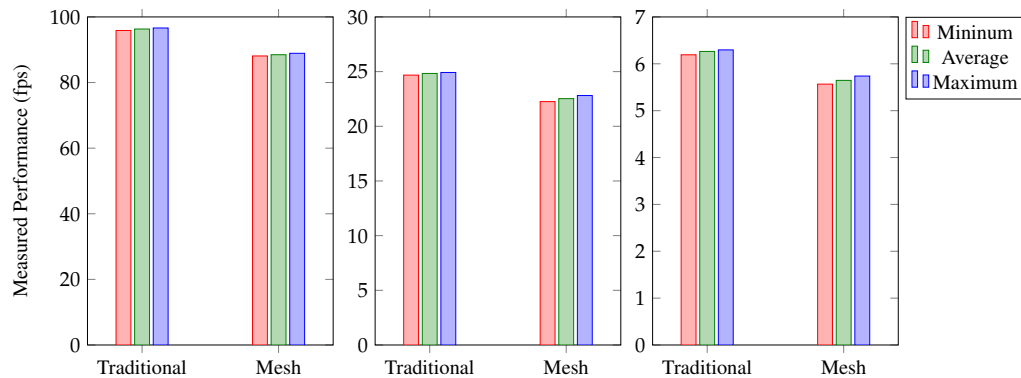


Figure 52: Measured performance of both pipelines when tessellating a quad with different tessellation levels: 2048 (left), 4096 (middle) and 8192 (right). Mesh shader limited to 64 tessellation levels.

With this acquired knowledge on how the mesh pipeline fares with tessellation, a project using tessellation shaders can be adapted to use the mesh pipeline instead, to compare how they fare in a real world implementation.

To chosen project renders ocean waves by applying transformations to vertices on an extensive tessellated grid. This grid is constituted by the 256 instances of sub-grids, each being tessellated by tessellation shaders.

In this project, the camera is placed above the ocean waves and directed towards the horizon line, as shown in Figure 53. Thus, in this scenario, frustum culling is done to discard the sub-grids outside of the camera's view and the tessellation levels of each sub-grid are defined by their distance from the camera (*i.e.*, the farther the sub-grid, the lower the tessellation applied is).

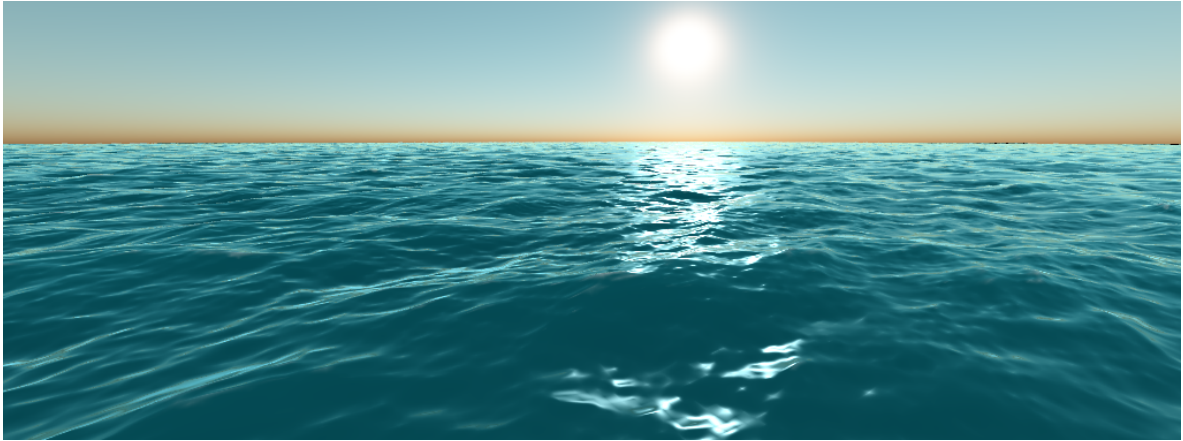


Figure 53: Render of the Ocean Waves project.

Once this project was translated to the mesh pipeline, the results were underwhelming, compared to those seen in Figure 50 and Figure 51. Figure 54 shows the these results for various configurations of the mesh pipeline, as well as the measured performance of the traditional one. These configurations differed in the `max_vertices` parameter, because increasing allows for higher tessellation levels, and levels of tessellation, with these ranging from the maximum level the tessellation shader can achieve (*i.e.*, 64 levels) to thirty-two times as much tessellation per sub-grid (*i.e.*, 2048 levels).

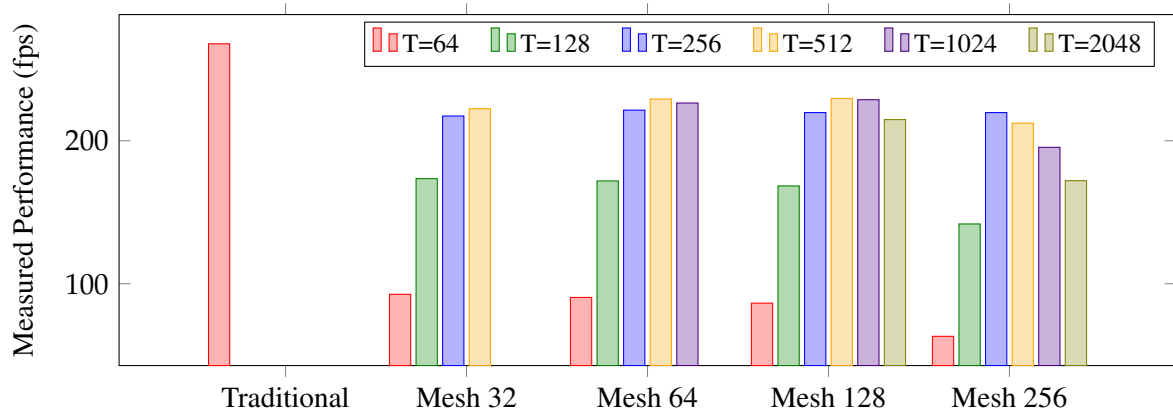


Figure 54: Recorded Performance of the Ocean Waves project on the both pipelines with several mesh pipeline configurations.

In none of these configurations was the mesh pipeline able to beat the traditional pipeline, or even match it. Whilst in the previous case the mesh pipeline performed slightly worse than the traditional, when the maximum amount of tessellation of both was the same, now the mesh pipeline performed ridiculously worse. Additionally, when increasing the amount of tessellation of each work group above 64, the performance significantly increases but isn't able to achieve the surpass the traditional. This contrasts significantly with the performance metrics shown in Figure 51.

This significant difference could be related to the added computation per-vertex that transforms the grid into a wavy ocean. To test this, a heavy per-vertex workload was added to the project of Figure 51, and the results shown in Figure 55 speak for themselves.

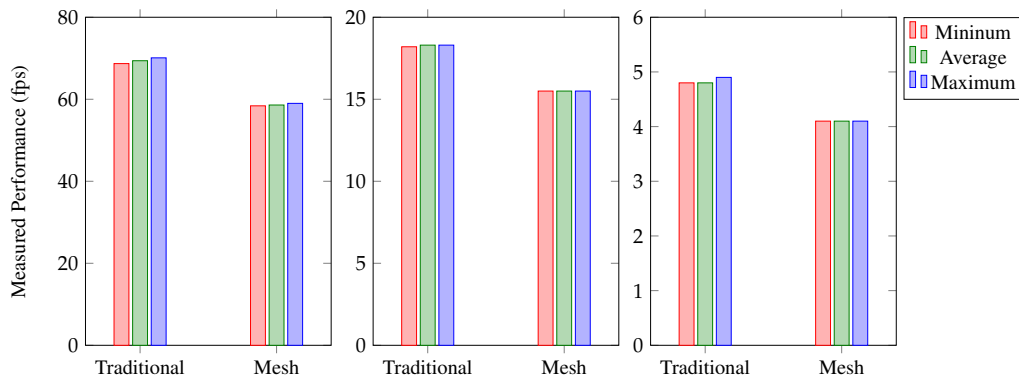


Figure 55: Measured performance of both pipelines when tessellating a quad with different tessellation levels: 2048 (left), 4096 (middle) and 8192 (right). Added heavy per-vertex workload.

In the same scenarios in which previously the mesh pipeline was severely outperforming the traditional pipeline, once a per-vertex workload was added to both pipelines, the traditional pipelines maintains a comfortable lead over the mesh one.

Thus, it seems that the mesh pipeline is able to outperform the tessellation shaders in vertex emission, by being able to achieve a higher tessellation level per shader, but once significant per-vertex computation is added, the tessellation shaders remain the faster way of dynamically tessellate a quadrilateral.

4.10 PROCEDURAL GENERATION WITH THE MESH SHADER PIPELINE

For the final test between the pipelines, the role of the Geometry shader will be taken by the Mesh shader. Even though Geometry shaders aren't particularly efficient, sometimes their use couldn't be avoided, as their ability to generate any kind of geometry on the fly was previously unmatched.

However, with Mesh shaders, much of the same functionality is given. To tackle this test of comparing the use of a mesh pipeline instead of a traditional pipeline with geometry shaders, the task of procedurally generating grass blades will be tested.

To do this, a single vertex is transformed into a grass blade composed of 5 vertices and 3 primitives. As a single grass isn't demanding enough for a test, for the first test 65000 instances of the traditional pipeline were executed and 65000 work groups of Mesh shaders were launched.

As can be seen in the left bar graph of Figure 56, in this scenario both pipelines perform about the same. The traditional pipeline registered on average one to two FPS higher, but said differences in for a performance this high can be attributed to margin of error.

For the second test, the number of grass blades per instance was increased up to the limit a Geometry shader can emit. The amount of geometry that a Geometry shader can produce is limited by the `GL_MAX_GEOMETRY_OUTPUT_VERTICES` *OpenGL* property, which is set to 85, equating to 17 possible grass blades per Geometry shader instance. Thus, the middle bar graph shows the measured performance with 65000 Geometry and Mesh shaders, both emitting 17 grass blades per instance.

In this scenario, the mesh pipeline takes the lead, being able to output about 40 additional FPS, which equates to about 12.5% performance gain.

As for the last test, the number of grass blades per Mesh work group was increased to the maximum achievable. Each Mesh shader can output a maximum of 256 vertices, which equates to 51 grass blades composed of 5 vertices. As the Geometry shader is limited to 17 grass blades per instance, the number of instances was tripled, as $3 * 17 = 51$, in order to match the total number of blades produced by both pipelines.

In this final scenario, the mesh pipeline achieves an even greater performance advantage, as can be seen in the right bar graph of Figure 56. In this case, the measured performance of the mesh pipeline was almost 50% higher than the one of the traditional pipeline.

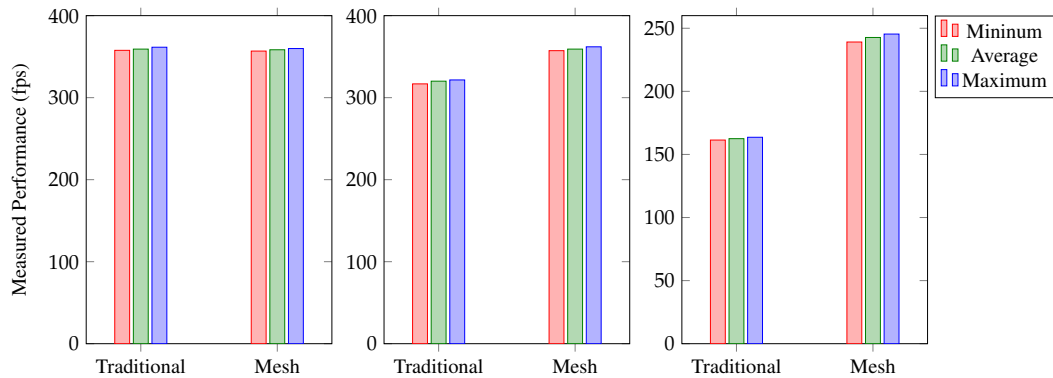


Figure 56: Measured performance of both pipelines when generating different amounts of grass blades: 65000 (left), 1105000 (middle) and 3315000 (right). For each scenario, different amounts of grass blades per shader instance are emitted.

These results, however, can only be achieved by following the fundamentals learned in Section 4.4 when setting up the mesh pipeline. If the tests with 1105000 and 3315000 total grass blades are executed with a single blade of grass per shader, instead of maximum number of blades possible, the results obtained become quite different, as shown in Figure 57.

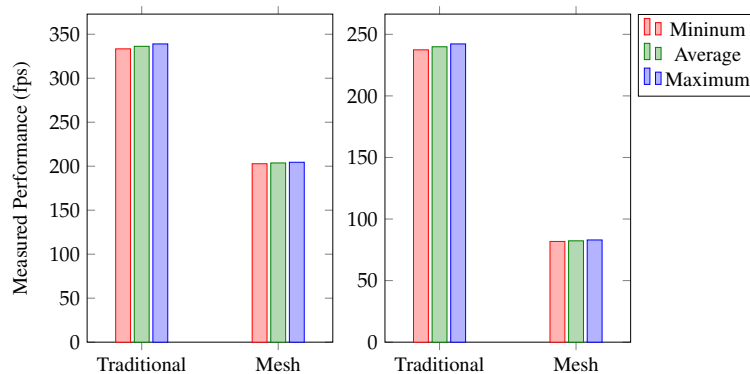


Figure 57: Measured performance of both pipelines when generating different amounts of grass blades: 1105000 (left) and 3315000 (right). For each scenario, a single grass blade per shader instance is emitted.

In this case, the fact that geometry shader isn't emitting 17 grass blades per instance seems to benefit the performance of the traditional pipeline. Inversely, the mesh pipeline, which is capable of emitting up to 51 grass blades per work group, immensely drops its performance when a single blade is emitted per work group.

From this change alone (*i.e.*, dropping the amount of grass blades per shader from 17 and 51 to 1) and the introduction of a Task shader, which was needed to amplify the number of work groups launched above

65535, the mesh pipeline turned a confident performance advantage over the traditional pipeline into a huge performance loss.

As such, an early conclusion that could be taken from these tests, is that the mesh pipeline can generate geometry slightly more efficiently than the traditional pipeline can with geometry shaders, if used wisely.

However, as seen in [Section 4.9](#), a single type of tests might imply a performance gain that might not be applied in all scenarios. Thus, even though these tests imply a better performance of the mesh pipeline compared to the traditional pipeline with geometry shaders, this could not be the case for every geometry shader use case.

4.11 SUMMARY

The tests made in this chapter showed that Mesh and Task shaders can provide better performance than the traditional pipeline in specific scenarios and conditions. Thus, these shaders aren't a revolution in the sense that the traditional pipeline should be discarded and replaced with the mesh pipeline. In fact, these shaders are merely an alternative approach, which should be used when its compute paradigm allows for resolutions not achievable with the traditional pipeline.

CONCLUSIONS AND FUTURE WORK

This chapter explores the conclusions that can be made with the work accomplished with this thesis. Additionally, it provides an insight on the possible avenues not explored in this thesis that would serve well as continuation to the investigation on the value of Mesh shaders.

5.1 CONCLUSIONS

The introduction of Mesh and Task shaders into the pool of shaders which developers can use to accomplish their tasks is undoubtedly positive. The flexibility of Mesh and Task shaders should allow developers to explore different approaches to solving their problems, which is certainly desired.

However, Mesh and Task shaders don't appear to solve a major problem with the traditional method of rendering. Compute shaders already enabled general-purpose parallel computing with the GPU and Vertex, Tessellation and Geometry shaders already allowed for the geometric processing facilitated by Mesh shaders. Thus, Mesh shaders mostly appear to eliminate the bottleneck of having to store the results of a Compute shader's computation in memory before it can be passed on to a rendering pipeline.

As such, the value of Mesh and Task shaders should be measured in the performance gain they offer over the traditional pipeline. This is, however, a complicated thing to accomplish, as the traditional pipeline has gone through several optimizations over the years and is in a very mature state.

In this thesis, it was possible to find scenarios where the mesh pipeline was able to outperform the traditional pipeline, but said scenarios also proved the volatility inherent to these shaders. For example, in the case of tessellation, the Mesh shaders were able to outperform the Tessellation shaders when applying significant amounts of tessellation to a grid. However, once per-vertex computation was added to the equation, the traditional pipeline suffered no performance decline, whilst the mesh pipeline declined to performance values below those obtained with the traditional pipeline.

Another interesting case of study is the rendering of 3D models. In some cases, the performance obtained with a mesh pipeline was able to surpass that of the traditional pipeline, but in a similar amount of times it performed worse. This variance in results is worsened by the fact that to achieve said results several configurations of Mesh shaders needed to be tested to figure out which fared better with each particular model.

As for procedural generation with Geometry shaders, the Mesh shaders were able to obtain better performance metrics when emitting a higher amount of geometry per shader instance. However, when both shaders were set to the same small amount of geometry per shader, the Geometry shaders outperformed immensely the Mesh shaders, as the latter shaders suffered significantly from the reduction in geometry emitted per instance.

This lack of clear-cut performance gains right out of the gate, aggravated by the additional effort needed, implies that these shaders are not the solution for most type of developers. However, experienced developers looking to optimize every step of their rendering process should find the flexibility provided by Mesh shaders encouraging, as there's potential for significant performance gains, even at the cost of additional labour.

In conclusion, Mesh and Task shaders are not meant as replacement of the geometric portion the traditional pipeline, but rather as an alternative for advanced users.

5.2 PROSPECT FOR FUTURE WORK

During the development of this thesis, not every possible avenue could be explored. Given the limited time for this development, some aspects couldn't be explored in depth. As such, this section outlines this possible avenues, as they could benefit the final performance of the mesh pipeline.

Textures vs SSBOs

When exploring the rendering of 3D models with Mesh shaders, a storage mechanism had to be chosen for the models' data. In this thesis, SSBOs were used but different storage mechanisms (e.g., textures) are available. As each mechanism was not tested, this thesis can't ensure that the mechanism chosen is the best option for use with Mesh shaders.

Buffer arrangements

As for the data representation for the 3D models, the approach suggested by NVIDIA was used. The theoretical benefits of said approach over the traditional simpler approach are clear, but experiments could have been made to measure said benefits and perhaps other buffer arrangements could have been found.

Meshlet Distributor

During this thesis, a translator was developed to sequentially assign the primitives of a Wavefront model to meshlets. However, this doesn't ensure that the meshlets make use of vertex reuse, as that's entirely dependent on the original model's reuse. That's why it's referred to as a translator, as it merely translates from the models' format to the buffer format.

Even though this was the strategy used, a better approach would be to factor in vertex reuse in the meshlet creation process, so that every meshlet can emit as much primitives as possible for a set maximum amount of vertices. As seen in Section 4.4, reducing the number of meshlets emitted has a great effect on the performance and by enhancing the vertex reuse of a model's meshlets, a lesser amount of them is needed.

However, designing this distributor would require a significant amount of time. Instead, several other primitive reordering algorithms were tested (in Section 4.3), but these were designed for the traditional pipeline and not the meshlets paradigm.

Wider range of tests

Testing the performance of a rendering pipeline is not a small feat. In this thesis, the pipelines were tested for the rendering of 3D models, tessellation and procedural generation. Nevertheless, there are numerous possible applications that could be better suited for one or the other.

BIBLIOGRAPHY

- Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition*. A. K. Peters, Ltd., USA, 4th edition, 2018. ISBN 0134997832.
- Antnio Ramires Fernandes. Nau 3d engine, 2014. URL <https://github.com/Nau3D/nau>.
- Glenn Fiedler. Tessellating the go stone | gaffer on games. https://gafferongames.com/post/tessellating_the_go_stone, 2013. [Online; accessed 26-January-2021].
- Tom Forsyth. Linear-speed vertex cache optimisation. https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html, 2006.
- Atsushi Haneda. About g-buffers in games. <https://hanecci.hatenadiary.org/entry/20130818/p1>, 2013. [Online; accessed 27-January-2021].
- Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, page 269–276, USA, 1999. ACM Press/Addison-Wesley Publishing Co. ISBN 0201485605. doi: 10.1145/311535.311565. URL <https://doi.org/10.1145/311535.311565>.
- Christoph Kubisch. Introduction to turing mesh shaders. <https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/>, September 2018.
- Morgan McGuire. Computer graphics archive, July 2017. URL <https://casual-effects.com/data>.
- Patrick Mours. Mesh shaders in turing. <https://on-demand.gputechconf.com/gtc-eu/2018/pdf/e8515-mesh-shaders-in-turing.pdf>, 2018.
- NVIDIA. The world's first gpu. <https://web.archive.org/web/20040414145655/http://www.nvidia.com/page/geforce256.html>.
- Nvidia. Cuda zone. <https://developer.nvidia.com/CUDA-zone>, 2017. [Online; accessed 18-January-2021].
- OpenGL Wiki. Compute shader. https://www.khronos.org/opengl/wiki/Compute_Shader, 2019. [Online; accessed 28-January-2021].
- University Computer Graphics Laboratory Stanford. Stanford 3d scanning repository, August 2014. URL <http://graphics.stanford.edu/data/3Dscanrep/>.
- Chris Vuong and Long Pham. Gpu architecture. <http://meseec.ce.rit.edu/551-projects/fall2016/2-3.pdf>, 2016. [Online; accessed 18-January-2021].

- Wikipedia. Graphics processing unit — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Graphics_processing_unit&oldid=1003575847, 2021a. [Online; accessed 31-January-2021].
- Wikipedia. Tesla (microarchitecture) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Tesla_\(microarchitecture\)&oldid=1000062015](https://en.wikipedia.org/w/index.php?title=Tesla_(microarchitecture)&oldid=1000062015), 2021b. [Online; accessed 17-January-2021].

