Enhancing Grass Rendering Through Mesh Shaders

Alessandro Sportelli

(BSc) Computer Games Applications Development, 2026

School of Design and Informatics

Abertay University

Table of Contents

Table of Figures

Table of Tables

Acknowledgements

I would like to thank my supervisor Erin Hughes for her guidance and enthusiasm throughout the project which this dissertation is based on, it has been vitally helpful and I’m very grateful to have had such an inspiring supervisor.

Abstract

Mesh shaders are a relatively new and novel way to process geometry for rendering. They have several new features which traditional vertex, geometry, and tessellation shaders do not – mainly that they can both add and more crucially remove vertices from any geometry which they are processing. This feature opens the door for dynamic levels of detail in processed geometry which could help improve visual fidelity and performance when rendering.

This dissertation aims to evaluate the performance and viability of dynamic levels of detail for grass rendering, comparing a traditional approach with discrete levels of detail to a new mesh shader approach featuring dynamic levels of detail.

An application was developed with C++ and the Vulkan graphics API which features both a traditional grass rendering pipeline with discrete levels of detail and a modern mesh shader approach to grass rendering with dynamic continuous levels of detail.

Results gathered from the application showcase the effectiveness of both pipelines in a variety of scenarios evaluating different distances for the high level of detail to cover with varying grass blade counts. The associated metrics help showcase the strengths and weaknesses of each pipeline and allow conclusions to be drawn of the viability of mesh shaders and continuous LODs for grass rendering in real-time applications.

Analysis of the results found that in scenarios at close to medium distances with grass blade counts of 240,000 and below the mesh shader based pipeline was able to outperform a traditional approach, beyond these scenarios however the traditional vertex shader based pipeline with discrete levels of detail outperforms the mesh shader approach – particularly at high distances with very large grass blade counts.

Abbreviations

GPU – Graphics Processing Unit

CPU – Central Processing Unit

LOD – Level of detail

LODs – Levels of detail

NS – Nano Seconds

SM – Streaming Multiprocessor

API – Application Programming Interface

CSV – Comma-Separated Values

Chapter 1 – Introduction

With video games increasing in scope year on year and the never-ending strive for realism pushing ever forward one often overlooked and yet vitally important part to creating photorealistic worlds is foliage, and more specifically grass. With grass typically always covering a significant portion of the player's screen it is incredibly important for games which strive for realism to invest significant time into creating a high-fidelity grass system capable of maintaining a realistic look while under careful scrutiny from always being within the player's vision. The main goals of a modern grass system are typically primarily to create a digital approximation of realistic looking grass then secondly, and arguably more importantly for the sake of the player experience, to make that grass render incredibly efficiently. With grass fields potentially being made up of millions of individual grass blades it is absolutely vital to ensure that the rendering time for a single grass blade is as low as possible. To that end several optimisation techniques are regularly employed to reduce the amount of computation required by the CPU and the GPU.

One widely used technique for optimisation is varying the level of detail for the model which is being rendered, ensuring in close ups the model is rendered at a high level of detail however, as it pans away from the camera it switches to an alternate version of itself with less detail – potentially switching multiple times at increasingly further distances. This is an excellent method to save on computation as the decrease in detail can often go completely unnoticed by users due to the distance from the camera and rendering less detail can save significantly on computation, particularly in scenarios where it can be applied to many rendered models simultaneously such as with trees, foliage, or blades of grass.

With the recent introduction of mesh shaders, the possibility has opened up to create a model's levels of detail in a novel way. Previously the shaders available to developers could only be utilized to alter the positions of a processed models' vertices, or through some very expensive computations, add in vertices. Mesh shaders enable developers to increase the efficiency of creating new vertices and uniquely also allow developers to remove vertices from processed geometry, this opens the door for a new method of LOD creation where instead of having multiple

saved versions of a model – a single high detail version can be processed and dynamically have its detail lowered to the required level through the shader.

1.1 Aim

This dissertation discusses a project which aims to develop an implementation of a modern geometry based grass rendering system making use of both a traditional vertex shader based pipeline featuring a typical discrete LOD system and a modern mesh shader based pipeline featuring a more advanced continuous LOD system. Data will then be gathered on the performance of both pipelines to compare and evaluate the effectiveness of the mesh shader based pipeline over the traditional vertex shader based pipeline. From this data a conclusion will be drawn on the viability of the use of mesh shaders for grass rendering, and the effectiveness of a continuous LOD system over a traditional discrete LOD system.

1.2 Research Question

This dissertation is looking to answer the question: “Can Mesh / Task shaders can be used to improve performance of grass rendering over traditional methods through the use of continuous LODs.”

1.3 Chapter Overview

Chapter 2 will discuss relevant literature about the traditional graphics pipeline, highlighting potential weaknesses. Mesh shaders will also be discussed as well as grass geometry and certain other key concepts which will be relevant throughout the further chapters. Chapter 3 will discuss development and implementation details, highlighting methods used to test performance and gather valid data on the created application. Chapter 4 will then present the gathered data which will then be discussed and analysed within chapter 5. Chapter 6 will then draw conclusions and present opportunities for further research.

Chapter 2 - Literature Review

In order to correctly and successfully carry out this project it was vitally important to ensure significant background literature be read on traditional graphics pipelines as well as the modern mesh and task shader variations. Additionally, it was vital to explore the general techniques which can be used to enhance and optimise grass rendering. This reading gave significant insight into how to best approach the project and research.

2.1 The Programmable Pipeline

The programmable pipeline refers to modern graphics pipelines which feature several stages that modern GPUs allow the developer to completely alter through custom shader programs. Prior to the advent of programmable shader stages, developers had been stuck using fixed function pipelines which severely limited their scope and potential due to them only having certain provided (and fixed, hence the name) functions to choose from for pixel or vertex manipulation.

Starting with graphics cards released in 2001 developers have been able to program several stages of the pipeline, commencing with the pixel shader stage soon followed by the vertex shader and then several others.

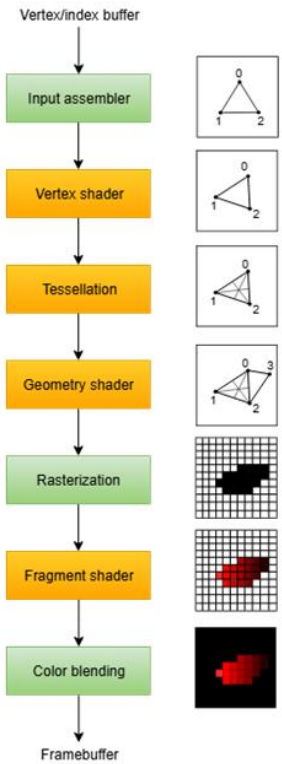

Figure 1, Traditional programmable pipeline layout. Khronos group.

2.2 Geometry Processing

The pipeline starts processing geometry at the input assembler where the provided geometry data gets constructed into a sequence of vertices which will be passed through and processed by the vertex shader.
Within the vertex shader developers are able to “directly control the operations the GPU performs for each vertex” (Maughan C, Wloka M, 2001). The shader gets called once per vertex, supplying the appropriate vertex data as input which can then be manipulated through “almost arbitrary computations per vertex” (Ebner M. Reinhardt M. Albert J. 2005).
Following the vertex shader is an optional stage called the tessellation shader. When in use, this shader stage allows the previously processed geometry from the vertex shader to be subdivided – creating more vertices between the ones which were supplied. This can be particularly useful to add detail to models based on their distance to the camera as a more primitive form of a continuous level of detail system.
Unfortunately, the tessellation shader comes with certain limitations; vertices can only be added following certain built-in patterns, and no vertices can be removed.
A second, optional stage, called the geometry shader occurs after the tessellation shader or the vertex shader should a tessellation shader not be in use.
The geometry shader takes in a single primitive (triangle, point, or line) at a time and can be used similarly to the vertex shader to transform the vertices of the supplied primitive. However, unlike the vertex shader the geometry shader can be used to create and subsequently manipulate additional vertices. This enables developers to potentially create multiple primitives from a single primitive input, which can be particularly useful for particle systems or for cases where large amounts of low detail geometry is required such as grass or fur rendering.
Despite this the geometry shader rarely sees use due to it being extremely slow when generating geometry, with it almost always being faster to simply send the required geometry directly to the GPU instead of generating it through the shader as shown by Patidar S. et al. (2007, p.5). As a result of this you will almost never find a geometry shader in a production environment as they are simply too slow for any meaningful application.

2.3 Rasterization

Rasterization is a fixed function stage in the rendering pipeline at which the previously processed geometry gets processed into fragments. Fragments are pixels stored within a 2-dimensional image containing not just colour information but also additional, associated data such as depth values (Vulkan Documentation, 2026), at this stage any per vertex values also get interpolated across each primitive (Microsoft, 2020). Rasterization operates per primitive and typically consists of determining which squares of a pixel grid are occupied by the primitive. Depth values then get assigned to each square and they are then passed on to the fragment shader. Any fragments which lie outside of the framebuffer or simply outside of the screen get discarded.

2.4 Fragment Shader

Once rasterization has taken place the final programmable stage of the pipeline occurs: the fragment shader. The fragment shader executes once per processed fragment and is typically where things such as lighting calculations or surface shading would take place. It is the shader which controls how the surface of the geometry being rendered will look. Due to the fragment shader being invoked once per fragment as opposed to once per pixel occasionally multiple fragments will attempt to draw to the same pixel. In this case whichever fragment is closest to the camera will be visible however all other fragments will still be processed as if they were visible. This issue gets referred to as overdraw and can commonly occur in scenes with many models densely packed together such as with fields of grass.

2.5 Instancing

Instancing is a technique in rendering where multiple copies of the same geometry can be rendered with a single draw call (Wikipedia, 2025). To submit geometry to the GPU for rendering is a slow and costly operation and as such minimising how many draw calls are used is vital to keeping high frame rates. Instancing enables the rendering of areas like forests or grass fields which often utilise a handful of repeated models hundreds or thousands of times without clogging up the CPU with endless

repetitions of identical draw calls. Instead by submitting each model once with a single draw call containing a parameter specifying the number of instances desired, hundreds or thousands of copies can be rendered while only passing the geometry though a single time (Carucci F, 2005). Each individual instance of the geometry can then be customized through the programmable shader stages in the pipeline with per instance variance through use of a unique instance ID (GLSL Documentation, 2017).

2.6 Frustum Culling

A frustum refers to a collection of six planes which define the field in which the camera can see. Typically for perspective cameras the view frustum will be a truncated pyramid with its tip being the cameras near plane and its base as the cameras far plane (Assarsoon U. Möller T, 1999). The view frustum defines what is visible by the camera and as such anything out with the view frustum will not be rasterised. However, geometry out with the view frustum will often still be processed as though it was visible, therefore in order to save on resources and improve draw times developers employ a system called frustum culling.
Frustum culling consists of checking whether geometry will lay within the view frustum, this typically occurs before the geometry itself has been processed, therefore if it does not lie within the frustum it can be discarded and no unnecessary computation will take place for it (Sunar M S. Zin A M. Sembok T M T, 2008).

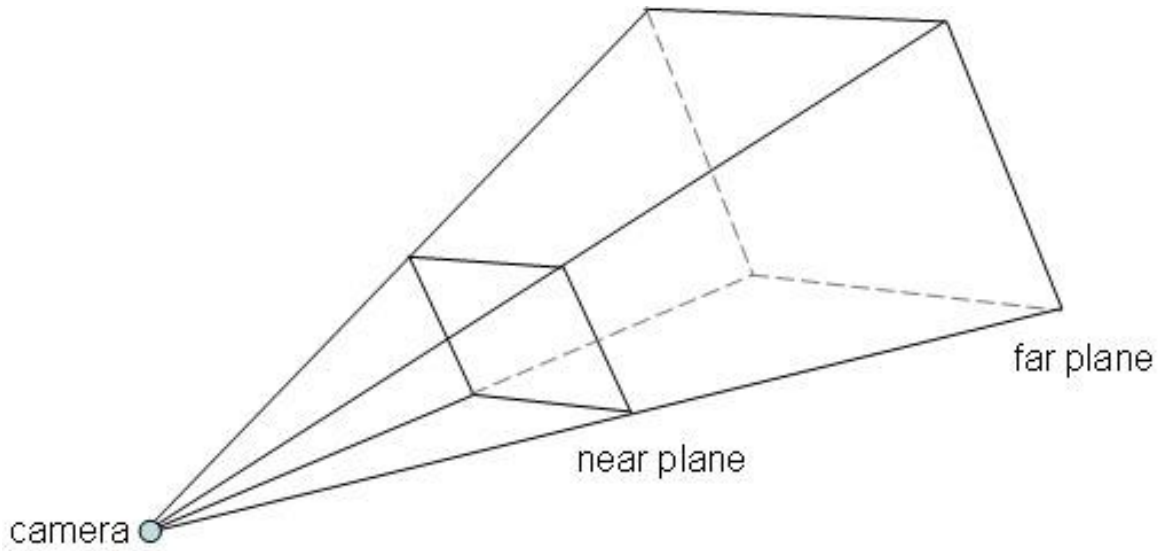

Figure 2, Visualisation of camera view frustum. ANSYS.

2.7 Level of Detail Systems

A level of detail system is an optimisation system typically used in real-time rendering where, rather than having a single full detail version of a model to be used, multiple versions of the model are instead created with diminishing levels of detail. Then, when rendering the model, a version with appropriate detail based on the distance the model is from the camera can be selected to be rendered as opposed to simply using the full detail model at all distances.

Typically, the highest detail version will be used close to the camera - with one or more other versions being rendered instead at further distances with diminishing quality as distance increases. This is a particularly effective method of optimisation as, with the model being at a meaningful level of distance from the camera it can often be seen that lower detail models can be almost indistinguishable from their higher detail counterparts.

In some extreme cases for objects such as foliage or trees at very far distances it can even be possible to replace the model entirely with a textured billboard, simply giving the appearance that the model is still there in the scene.

Determining which LOD to use can also be further optimised by offloading the work to the GPU through use of indirect draw calls. Indirect draw calls are draw calls that determine their parameters (in this case instance count) through data calculated and stored by the GPU. This data is often calculated through the use of compute shaders which are simply programs which operate on the GPU.

2.8 Mesh Shaders

Mesh shaders are a new way to process models when rendering, completely bypassing the traditional route of processing geometry. Mesh shaders were introduced with the main goal to “increase flexibility and performance of the geometry pipeline” (Hargreaves S, 2019). First introduced to directX12 in 2019 followed after by Vulkan in 2022, they are an alternative and novel method to process vertices through an almost compute shader like interface as suggested by Oberberger M. et al. (2023). Combining most aspects of vertex and geometry shaders, mesh shaders allow the opportunity to introduce new vertices during processing; however, unlike a traditional pipeline, mesh shaders also allow

developers to discard vertices at will - giving increased freedom over how many vertices and primitives are output to the rasterizer and allowing for geometry to get pre-culled as pointed out by Kubisch C. in the 2018 article introducing Turing mesh shaders. The mesh shader also comes with an accompanying task shader which is an optional step executed before the mesh shader which can be used to compute and dispatch the required number of “tasks” which are required, where each task is simply an instance of the mesh shader. This allows for potentially up to millions of mesh shader instances to be called from a single task shader dispatch on the CPU. Unfortunately, due to their modern nature mesh shaders are not supported nearly as widely as the traditional pipeline is - with the lower bound of supported graphics cards according to Levien R. (2023) currently being the NVIDIA GTX 1650 released in 2019 or the AMD Radeon RX 6400 which was released in 2022. Despite this they have already found use within the games industry, being prominently featured within Alan Wake 2 which was released in 2023 as discussed by Jansson E. in a 2024 Digital Dragons talk.

2.9 Grass Geometry

Grass geometry can be modelled in several ways depending on the desired fidelity, performance, and technical limitations; historically it was common to see grass being rendered through what is known as billboarding however modern grass systems tend to feature fully geometric grass blades.

Billboarding is a technique where one or multiple flat primitives or quads can be textured to give the appearance of detailed grass while maintaining excellently high framerates due to the extremely simple geometry. Billboarded grass could even be given the appearance of motion through manipulating the positions of the billboard’s vertices, a technique covered by Pelzer K. in his chapter of the 2004 release GPU Gems. Modern games however tend to use fully geometric grass where each blade is individually modelled, often as a triangle strip. This level of detail in grass has only been possible in more recent times due to the increase in computational power that graphics hardware has attained and is often only chosen by games pursuing very high visual fidelity due to how demanding the geometry is to process and render.

Due to the blades being modelled as a triangle strip LODs can be easily generated through simply having less triangles in the strip, or alternatively due to the simple geometry LODs can be generated at runtime through different shader approaches such as with the tessellation shader or as will be later discussed in chapter 3, with the mesh shader. The grass blades can be made to look very similar regardless of vertex count through having their vertices placed according to a Bezier curve, allowing for easier control of how the blade looks and curves as well as opening the door to easy animation through adjusting the Bezier control points – a technique showcased by Jahrmann K. and Wimmer M. (2017).

Chapter 3 – Methodology

This chapter will give an overview of the practical work which was undertaken to produce the artefact for this project. Descriptions of techniques and the justification for using them will be discussed to allow the method to be analysed and understood.

3.1 Overview

In order to answer the research question, a simple renderer was developed with the C++ programming language and making use of the Vulkan graphics API. This renderer was then utilized to build a simple scene to showcase the developed grass system, which was then in turn used to gather performance metrics on both pipelines with varying blade counts and LOD distances.
Gathered performance data was then analysed to evaluate the effectiveness of mesh shaders and continuous LODs for grass rendering to inform whether they could be more widely used for grass rendering in real time applications, or whether a traditional approach performs better.

3.2 Research and Preparation

Before development could commence, research had to be undertaken to understand how grass rendering is traditionally handled and what development had already taken place using mesh shaders. Additionally technical research had to take place in order to ensure all required features could be implemented within the renderer.

3.2.1 Grass Rendering

To better understand and approach the research question, in depth research was carried out to study how grass rendering has been traditionally implemented, how modern games render grass, and in which areas mesh shaders may be able to improve this process.
One of the key sources on modern grass rendering techniques was the 2022 Game Design Conference talk “Procedural Grass in ‘Ghost of Tsushima’” by Eric Wohllaib from Sucker Punch Productions. Within the talk several key techniques for a modern geometry based grass approach are discussed such as:

- Using individually modelled grass blades.

  Traditionally grass and foliage will typically be rendered through a technique called billboarding, in which one or several flat quads or tris will be textured to look like a clump of grass and then will either sit statically or rotate to face the camera depending on the approach. This technique can be seen in the chapter by Pelzer K. in the 2004 release GPU Gems. This technique is very effective as it can give the appearance of dense grass fields with very little computation required making it extremely effective on older machines, it is however limited by the same qualities which make it effective. The technique unfortunately lacks the ability to individually control grass blades, and its illusion can fall apart under careful scrutiny. With recent improvements to computational power developers are no longer as limited with scope and, as such, modern games (including Ghost of Tsushima) will render fields through individually modelled blades of grass. This allows developers to have control over fine details down to individual grass blades therefore more accurately emulating real fields of grass, even under careful scrutiny.

- Making use of grass clumping.

  Grass clumping is a technique where multiple blades belong to a single clump position to more realistically emulate how real grass grows. Using this approach allows for deeper control over grass blades as you can use the clump position to drive other parameters of the grass blades such as height, direction, and colour.

- Using a Bezier curve to place grass blade vertices.

  Having vertices follow a Bezier curve allows for easy and simple animation as, instead of having to manually animate each vertex, the curve’s control points can be tweaked and moved to animate all vertices in a controlled manner. Additionally, this allows for surface normals to be calculated extremely easily by getting the curve’s derivative and crossing it with a side vector.

3.2.2 Mesh Shaders

With mesh shaders being a relatively recent development, it was vitally important to research how they function and the best practices to follow for the project. Several sources were researched with a number of talks from the 2024 Digital Dragons conference being key resources to better understanding their usage, particularly that of Radosław Paszkowski which acted as a foundational starting point.
In addition to the general mesh shader resources, an article from GPUOpen on procedural grass rendering by Faber C. et al. (2024) which showcases an excellent simple approach to grass rendering through mesh shaders was referenced in depth throughout development.

3.3 Application Overview

When building a renderer, application design is vitally important to ensure things run efficiently and to reduce redundant work. The renderer built followed the framework outlined in the Vulkan introduction written by Alexander Overvoorde and made use of several helper libraries alongside the Vulkan API such as TinyObjLoader to handle the loading of different models, OpenGLMathematics (GLM) for help with various mathematical functions used in 3D graphics, and GLFW to handle the creation of the renderers window and to handle input from the keyboard and mouse. The ImGui library was also included to allow for a simple user interface to be added enabling the ability to tweak certain parameters about the applications grass, wind or general rendering settings in real-time while the application is running. Additionally, the NVIDIA Nsight Perf library was added to the project to display real-time performance information and allow performance reports to be generated, which is further expanded within section 3.5.1.
The developed renderer was then used to create the project's main program which would be used to measure performance and gather the necessary data for the research question.

3.3.1 LOD Vertex Counts

Within the application definitions of each level of detail had to be set. For the purposes of this application, it was decided that the high level of detail model would

consist of a blade featuring 32 vertices and the low level of detail model would consist of a blade with 6 vertices. The vertex shader based pipeline simply renders the high LOD model up until a distance equal to the max high LOD distance parameter, then beyond this distance the low LOD model is rendered. The mesh shader based pipeline uses these values as its maximum and minimum vertex counts for the pipeline's continuous LOD system. Where a blade's vertex count will be equal to high LOD vertex count when next to the camera and the vertex count will linearly decrease until the blade is at a distance equal to the max high LOD distance parameter, at which the blade's vertex count will be equal to the low LOD vertex count. Where high LOD and low LOD are referred to in further chapters, these are the vertex counts they represent.

3.4 Pipelines

Several pipelines were created within the program, they consist of traditional rendering pipelines, compute dispatch pipelines, and a task / mesh shader pipeline. These pipelines are used to render all geometry and dispatch work to the GPU.

3.4.1 Ground Pipeline

The pipeline for rendering ground is relatively simple, consisting of a vertex shader to scale up the ground plane model and move vertices vertically according to a sampled heightmap. This vertex shader is followed by a simple pixel shader to texture the ground plane according to a sampled texture and darken it for a computationally cheap and simple rough approximation of ambient occlusion.

3.4.2 Skybox Pipeline

The skybox pipeline is also rather simple, consisting again of a basic vertex shader to simply scale up the box model and set the correct texture coordinates for each face. This is followed by a pixel shader to texture each face according to a supplied cube map.

3.4.3 Grass Position Compute Pipeline

The grass position compute pipeline is dispatched once for each required grass clump when the program is started in order to calculate and store the clump positions which will be used by the grass pipelines.
The compute shader begins by calculating the current invocations x and z position in a uniformly distributed grid, these positions then have a random offset applied to produce a more natural seeming grid distribution. The clump position's UV is then calculated relative to the size of the ground plane and is then used to sample the ground plane's heightmap to calculate the clump position's Y component. The grass position's W component is used to store a rotation value sampled from a noise map. The complete position and UV are then stored within a storage buffer to then be accessed later by the grass rendering pipelines.

3.4.4 Grass Culling Compute Pipeline

The grass culling compute pipeline is called at the start of each frame to cull out any clump positions which are outside of the view frustum.
The pipeline consists of a single compute shader which extracts the view frustum planes and then checks each clump position for whether it lies within the frustum or out with it. If it lies within the frustum the distance from the camera to the position is calculated and is used to separate the positions into one of two arrays within a new culled position's storage buffer. If the distance to the position is below the threshold for low LOD it will be placed in the high LOD position's array otherwise it will be placed into the low LOD position's array. Additionally, the respective instance count variable is incremented 4 times as there are 4 grass blades per clump.
Processing the grass positions in this way significantly increases performance with no cost to the final render as it removes the need to process thousands of grass blades which would otherwise be off screen.

3.4.5 Vertex Shader Based Grass Pipeline

When the program runs it is using the vertex based grass pipeline by default. It consists of a standard geometry pipeline featuring a relatively complex vertex shader and a standard pixel shader.

Rendering grass through this pipeline is done with two separate draw calls, one for high LOD grass and one for low LOD grass. Both draw calls are indirect and read their instance count from the values previously calculated during the culling process. Both draw calls utilise the same pipeline and as such the first step in the vertex shader is to work out whether the vertices being processed are high LOD or low LOD. This is done using the gl_baseInstance parameter where the high LOD draw call will have a base instance of 0 and the low LOD draw call has a base instance equal to the amount of high instance grass blades.

Once this is done the appropriate clump position array can be accessed and is stored as the blade's base position. A random seed is then calculated using randomisation functions by Max Oberberger (2024) to be used to drive all of the randomised aspects of the grass blade. The next step is to calculate the strength of the grass blades lean, predominantly driven by a user controlled parameter and the wind strength sampled from a scrolling noise texture. A blade direction vector is then calculated; this vector is used as the direction the grass blade will face and is mainly based on the direction the wind is moving with some random variance for a more natural appearance. The blade's height then gets calculated, again predominantly based off a user controlled parameter with some slight variance for a more realistic appearance. The next step is to calculate the blade position offset as a completely random vector - the offset allows for multiple blades to originate from the same clump position while not directly occupying that same position.

With the blade offset calculated the blades Bezier curve control points can now be defined. There are 3 points for the curve:

P0 is defined as the clump position with the offset added.

P2 is defined as P0 + the blades height and displaced by the direction of the blade.

P1 is defined as a point between P0 and P2 moved perpendicular by the user controlled curve strength parameter.

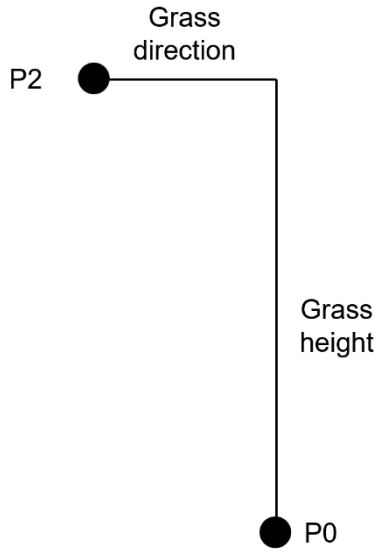

Figure 3, Depiction of P0 and P2

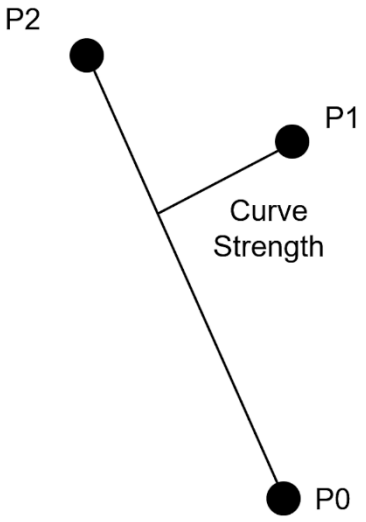

Figure 4, Depiction of P0, P1, and P2

A sideways offset is then calculated and applied to P0 and P1. This offset serves to give the blade width, tapering off towards the tip and is controlled via the exposed blade width parameter. The offset uses the ID of the vertex being processed to give itself either a positive or negative sign allowing for vertices to occupy both sides of the previously calculated Bezier curve.

The processed vertex is then placed along this curve according to its vertex ID and has its normal vector and texture coordinates calculated accordingly.

3.4.6 Mesh Shader Based Grass Pipeline

The mesh shader pipeline can be toggled on via a checkbox in the user interface. The mesh pipeline begins with a single dispatch command for the task shader where within the task shader, the high LOD instance count and the low LOD instance count are combined and used to calculate the correct number of mesh shader instances to launch.

Within the mesh shader the overall mesh shader work group ID is used to sample from either the high LOD clump positions or the low LOD clump positions - note that as the mesh shader LODs are calculated within the shader the positions can simply be sampled in order.

Once the clump position has been sampled the distance between the position and the camera is used to calculate how many vertices the blades at the clump should

have. A maximum and minimum vertex count are defined and the calculated distance is used to linearly interpolate between the two values. Following on, a loop is used to calculate vertices using the same methods as the vertex shader with a key difference of 4 vertices being calculated per thread; this then adds up to a full clump of 4 grass blades per mesh shader dispatch. Using the same methods as the vertex shader means the grass blades are visually indistinguishable and so when data is collected the same tests can be carried out for a fair comparison.

3.5 Data Collection

Testing was carried out on the application to evaluate the performance of both grass pipelines implemented. The testing was performed to gather data on which pipeline performs better in a variety of scenarios to ultimately conclude whether a mesh shader pipeline can outperform traditional methods of grass rendering.

3.5.1 Tools

In order to gather performance data on the rendering of the application the NVIDIA Nsight Perf SDK was integrated within the renderer to enable in-application data collection. Within the application a defined range was set to ensure gathered data pertained only to the rendering of the grass within the scene. The data output by the application was output to CSV files which were then analysed and collated to create data tables on each of the values to be analysed.

3.5.2 Method

In order to ensure a wide range of scenarios were tested within the application’s scene, a method for gathering data was developed which would evaluate the performance of the pipelines through several situations.

To evaluate the performance at varying blade counts, the camera was moved within the scene while facing directly towards the ground plane. At each distinct position the camera was moved to, a performance report was taken which would later be evaluated and compared.

Additionally, this test was repeated for various max high LOD distance values to ensure that testing evaluated not just the pipeline performance when rendering

various amounts of grass blades, but also the performance when rendering varying amounts of grass at each level of detail.

The camera heights used in testing are shown below in a table with the blade counts rendered at each position.

Blades Rendered	Camera Height
31168	50
111736	100
240976	150
418160	200
634868	250
871264	300

Table 1, Grass blades rendered at each camera height in testing.

These heights were tested for the following max high LOD distance values: 100, 150, 200, 250, 300.

To ensure data gathered was accurate and could be validated, testing was automated programmatically within the application. Timers were used to automate report generation and incrementation of the camera height, and max high LOD distance variables as follows:

Time elapsed (s)	Camera Height
0	50
10	100
15	150
20	200
25	250
30	300
35	50

Table 2, Elapsed time when camera moved to each height in testing.

Once the elapsed time reached 35 seconds the camera height resets to 50 as shown in the table, the timer then reset back to 0, and the max high LOD distance variable incremented by 50 until all reports had been generated.

Reports were automatically generated using a timer every 5 seconds, with the first one being generated after 4 seconds of application run time. These 4 seconds acted to give the programs performance some time to stabilise.

All reports were generated on a system with the following specifications:
Processor: AMD Ryzen 7 7800X3D
Memory: 32GB DDR5 6000MHz CL30
GPU: NVIDIA GeForce RTX 3060 (Locked to rated TDP)
Operating system: Windows 10 - 64 bit
GPU Driver: NVIDIA Game ready driver 580.97

The program scene was rendered within a 1291 pixel wide, 1026 pixel tall window.

3.5.3 Metrics

The reports contain information on an entire frame's metrics however a range was defined within the application to allow metrics relating to grass generation to be extracted separately, these are the metrics which were evaluated and compared in order to answer the research question.

4 distinct metrics have been collated into tables in order to evaluate and compare vertex and mesh pipeline performance. Note that in these metrics 'Throughput' refers to a percentage of the possible maximum rate that data can be processed on the specific hardware used in testing.

Metric	Reasoning
Duration (Ns)	To determine the overall time taken for the pipeline to execute.
L2 Cache Hit Rate (%)	To evaluate the effectiveness of memory access patterns and cache coherence.
SM Throughput (%)	To evaluate how close pipeline performance is to the theoretical maximum.
Warp Occupancy (Per SM) (%)	To give a sense of how utilised GPU cores are and evaluate use of parallelisation.

Table 3, Collected metrics and reasoning behind them.

Chapter 4 – Results

Data was collected on the performance of both the vertex shader based pipeline and the mesh shader based pipeline in the same environment and running on the same system.
Table 1 displayed in chapter 3.5.2 showcases the rendered grass blade count at the various heights used in performance testing and provides context to the figures represented within chapter 4.

4.1 Vertex Shader Based Pipeline Metrics

This section will present the metrics gathered for the vertex shader based pipeline. Tables with raw data are provided as well as graphs to visualise of the data gathered.

4.1.1 Duration

Table 4 showcases the duration of the processing time in nanoseconds required for the vertex shader based pipeline to render at each height used in testing and at each max high LOD distance measured. This data can also be viewed in figure 5 which showcases the gathered metrics as a graph.

The values below Max High LOD Distance refer to the values measured and the values below Height refer to the tested heights for each Max High LOD Distance.

VERTEX PIPELINE			**Duration (Ns)**			
Max High LOD Distance (world-space unit)			Height (world-space unit)			
	50	100	150	200	250	300
100	1.16E+06	1.19E+06	1.42E+06	1.81E+06	1.81E+06	2.72E+06
150	1.17E+06	1.20E+06	2.30E+06	1.88E+06	2.15E+06	2.71E+06
200	1.17E+06	1.19E+06	2.29E+06	3.60E+06	2.24E+06	2.68E+06
250	1.17E+06	1.19E+06	2.25E+06	3.80E+06	4.86E+06	2.79E+06
300	1.17E+06	1.20E+06	2.28E+06	3.77E+06	5.69E+06	6.19E+06

Table 4, Vertex pipeline duration values.

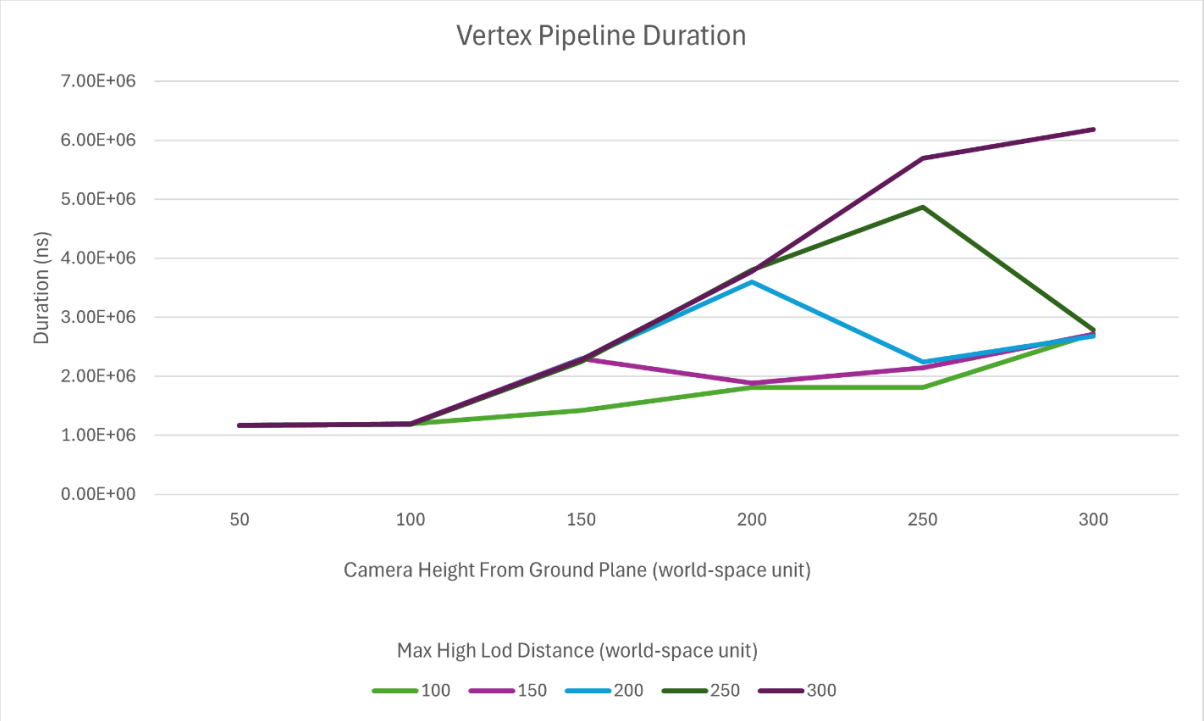

Figure 5, Vertex pipeline duration values.

4.1.2 Level 2 Cache hit rate

Table 5 showcases the level 2 cache hit rate as a percentage for the vertex shader based pipeline. This data is visualised as a graph in figure 6.

VERTEX PIPELINE	L2 Cache hit rate %					
Max High LOD Dist (world-space unit)		Height (world-space unit)				
	50	100	150	200	250	300
100	88.5892	86.0159	85.2789	86.727	89.6249	91.7529
150	87.9006	86.8868	92.6028	88.3673	90.4786	91.361
200	87.5044	86.9304	91.7107	94.9111	91.067	92.3941
250	87.3819	86.8886	93.3707	94.8259	94.9259	92.4789
300	87.4531	87.839	92.2947	94.5434	94.9459	95.9559

Table 5, Vertex pipeline L2 Cache hit rate values.

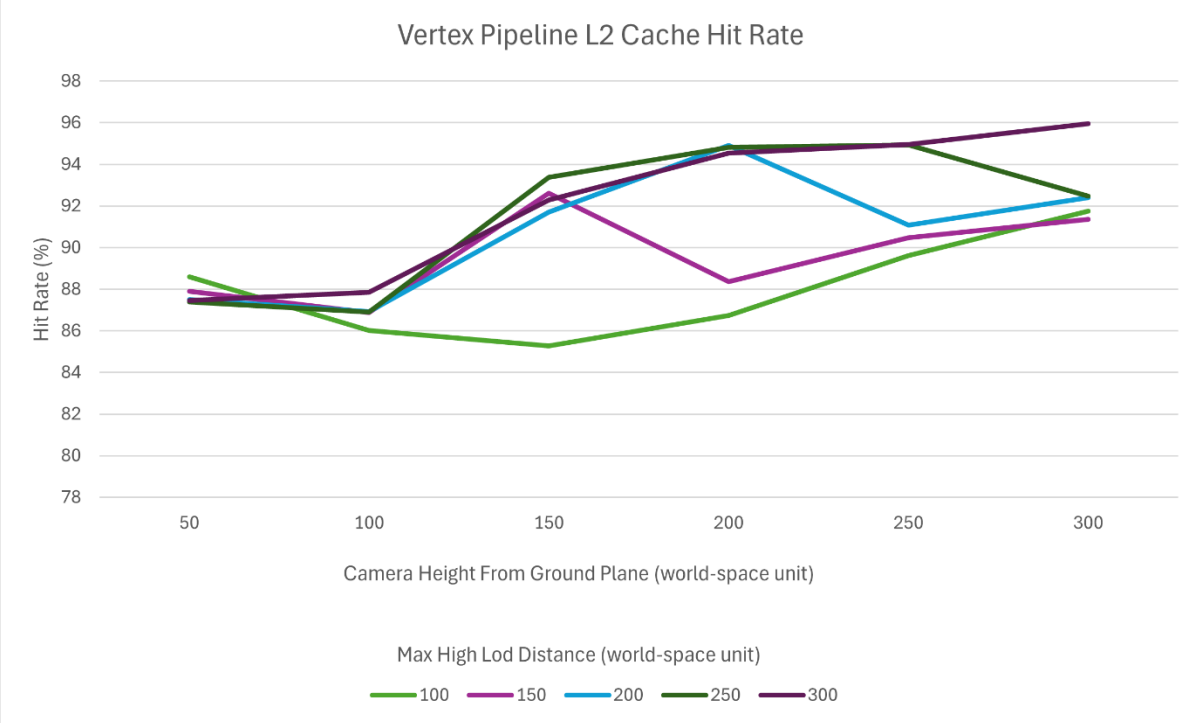

Figure 6, Vertex pipeline L2 Cache hit rate values.

4.1.3 Streaming Multiprocessor Throughput

Table 6 showcases the values gathered for the streaming multiprocessor throughput as a percentage of the theoretical maximum for the vertex shader based pipeline, these values are visualised within figure 7 as a graph.

VERTEX PIPELINE			**SM Throughput**			
Max High LOD Dist (world-space unit)			Height (world-space unit)			
	50	100	150	200	250	300
100	19.0049	16.1271	23.4916	36.6777	46.5184	53.191
150	18.8685	10.074	32.3686	36.9618	46.5706	53.2318
200	18.9382	10.0484	19.7862	49.0337	47.3687	52.6655
250	18.952	11.3811	19.3019	23.294	59.5687	53.6271
300	19.0442	11.4411	19.3524	19.7689	29.7222	63.147

Table 6, Vertex pipeline SM Throughput values.

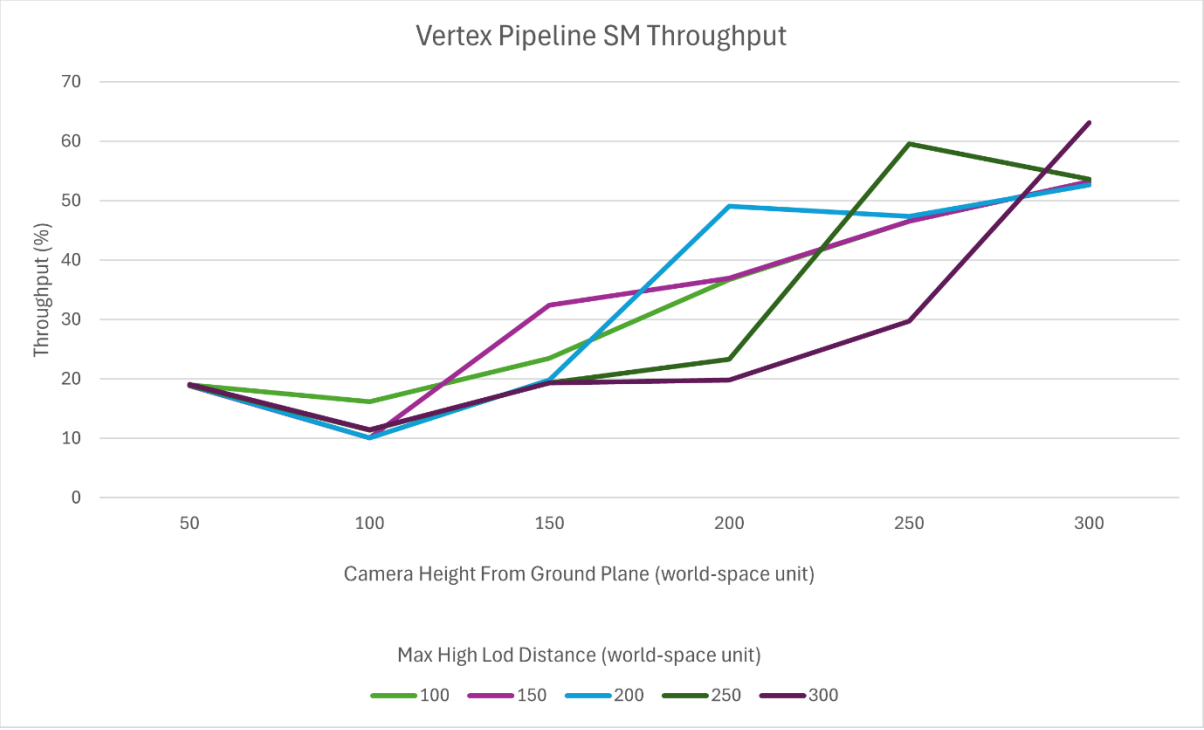

Figure 7, Vertex pipeline SM Throughput values.

4.1.4 Warp Occupancy (Per Streaming Multiprocessor)

Table 7 showcases the values gathered for warp occupancy (per streaming multiprocessor) as a percentage of the theoretical maximum for the vertex shader based pipeline. This data is visualised as a graph in figure 8.

VERTEX PIPELINE			**Warp Occupancy (%) (Per SM)**			
Max High LOD Dist (world-space unit)			Height (world-space unit)			
	50	100	150	200	250	300
100	9.32352	14.2217	22.249	33.7759	40.8935	41.1016
150	9.17117	13.123	21.7646	33.606	41.0793	41.0716
200	9.26177	13.1279	15.6719	32.4911	41.4739	41.1821
250	9.15998	13.052	15.2234	17.8021	40.1536	41.2655
300	9.31768	13.1455	15.181	15.7011	20.0109	40.029

Table 7, Vertex pipeline warp occupancy (Per SM) values.

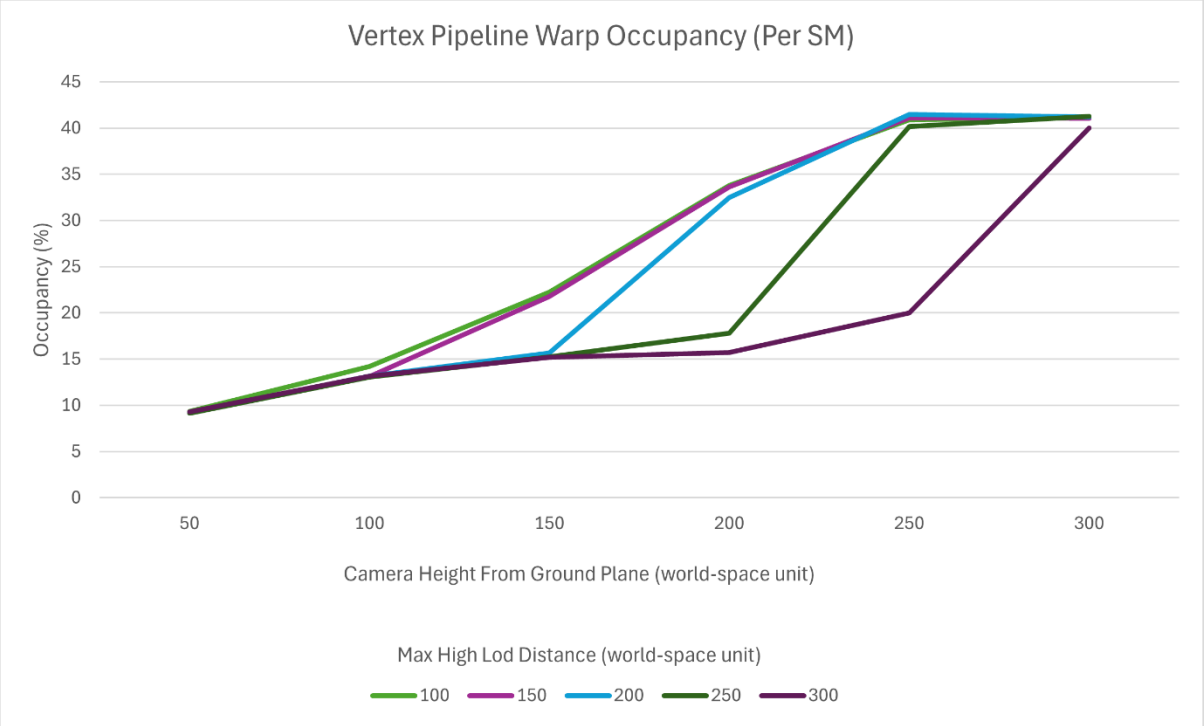

Figure 8, Vertex pipeline warp occupancy (Per SM) values.

4.2 Mesh Shader pipeline metrics

This section will cover the data gathered pertaining to the mesh shader based pipeline. Tables displaying raw data will be provided in addition to graphs visualising the gathered data.

4.2.1 Duration

Table 8 showcases the data gathered relating to the duration in nanoseconds of processing time required for the mesh shader based pipeline to render. Figure 9 showcases this data as a graph.

MESH PIPELINE			**Duration (Ns)**			
Max High LOD Dist (world-space unit)			Height (world-space unit)			
	50	100	150	200	250	300
100	9.93E+05	1.01E+06	1.40E+06	2.55E+06	4.33E+06	6.49E+06
150	1.04E+06	1.08E+06	1.68E+06	2.55E+06	4.34E+06	6.56E+06
200	1.09E+06	1.10E+06	1.90E+06	2.72E+06	4.38E+06	6.53E+06
250	1.09E+06	1.12E+06	2.11E+06	3.58E+06	4.42E+06	6.59E+06
300	1.09E+06	1.12E+06	2.26E+06	3.78E+06	5.77E+06	6.76E+06

Table 8, Mesh shader duration values.

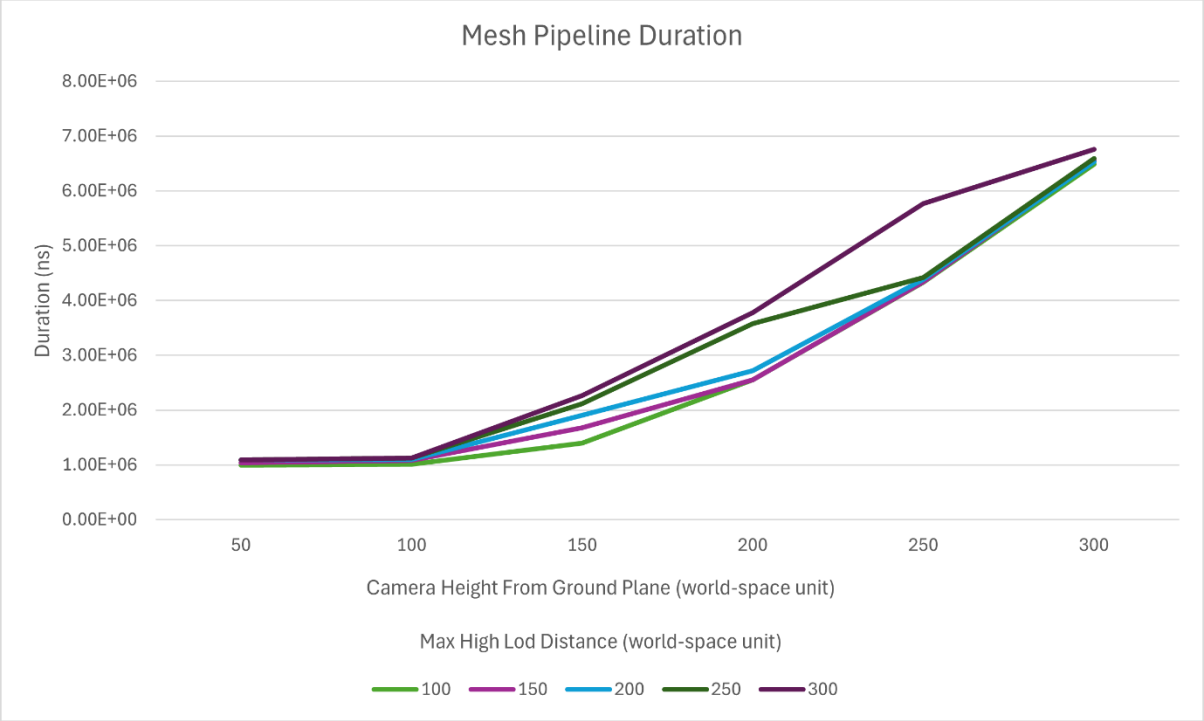

Figure 9, Mesh shader duration values.

4.2.2 Level 2 cache hit rate

Table 9 showcases the level 2 cache hit rate for the mesh shader based pipeline as a percentage. This data is visualised as a graph in figure 10.

MESH PIPELINE	**L2 Cache hit rate %**					
Max High LOD Dist (world-space unit)			Height (world-space unit)			
	50	100	150	200	250	300
100	81.6719	80.3793	82.6625	85.7335	89.338	89.5587
150	82.9058	82.6027	86.3145	84.6759	87.9782	93.5188
200	85.3501	84.9081	85.2394	88.3809	90.8085	87.3752
250	85.0704	80.93	89.375	88.601	88.5091	89.7907
300	85.6901	84.943	89.6843	90.6061	90.9005	91.8122

Table 9, Mesh shader L2 cache hit rate values.

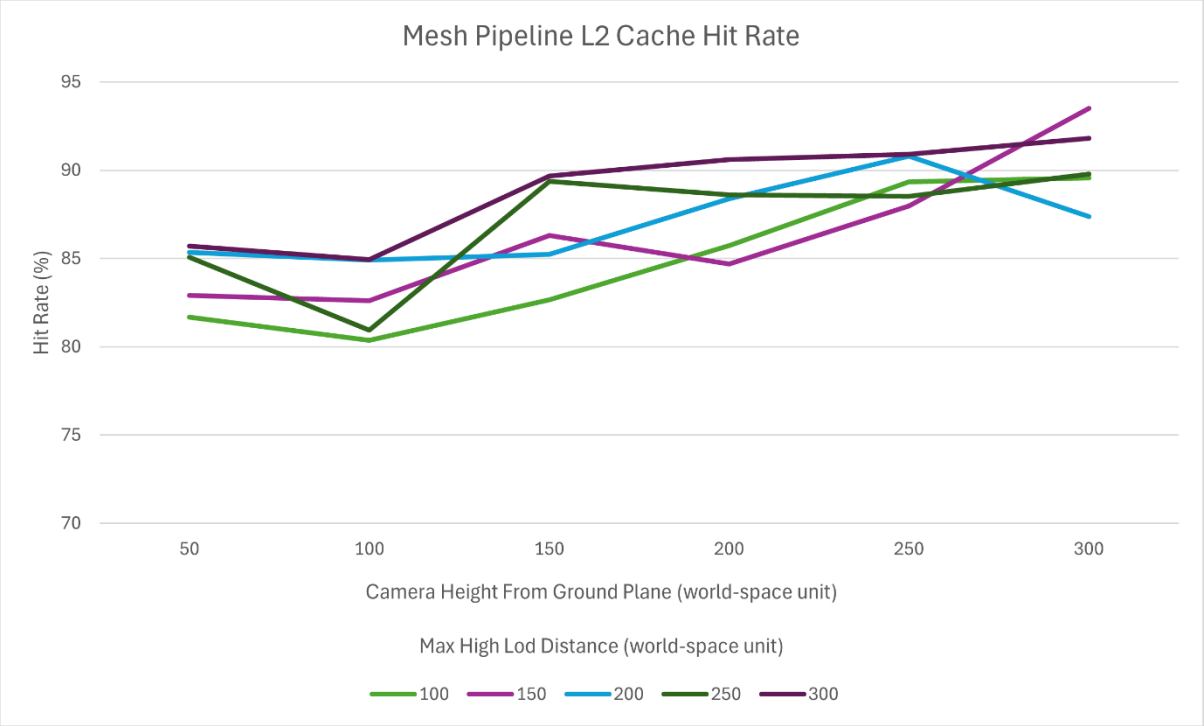

Figure 10, Mesh shader L2 cache hit rate values.

4.2.3 Streaming multiprocessor throughput

Table 10 showcases the mesh shader based pipeline's streaming multiprocessor throughput as a percentage of the theoretical maximum. This data is visualised as a graph in figure 11.

VERTEX PIPELINE			SM Throughput (%)			
Max High LOD Dist (world-space unit)			Height (world-space unit)			
	50	100	150	200	250	300
100	19.0049	16.1271	23.4916	36.6777	46.5184	53.191
150	18.8685	10.074	32.3686	36.9618	46.5706	53.2318
200	18.9382	10.0484	19.7862	49.0337	47.3687	52.6655
250	18.952	11.3811	19.3019	23.294	59.5687	53.6271
300	19.0442	11.4411	19.3524	19.7689	29.7222	63.147

Table 10, Mesh shader SM throughput values.

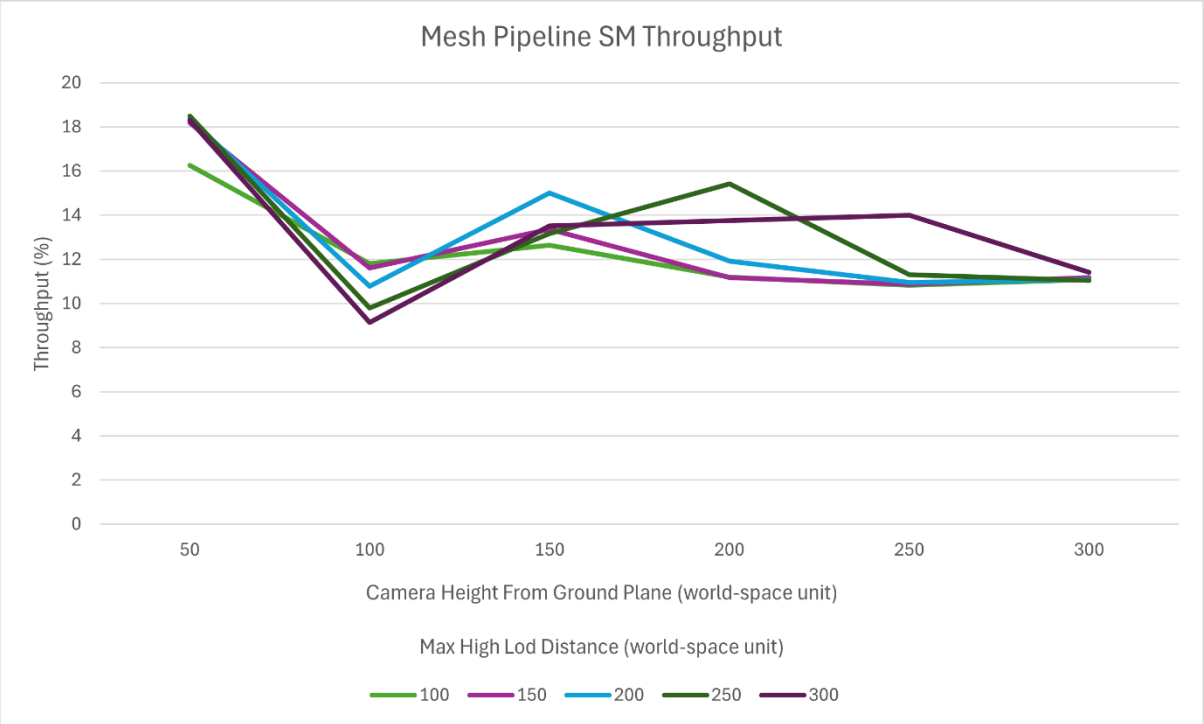

Figure 11, Mesh shader SM throughput values.

4.2.4 Warp occupancy (per streaming multiprocessor)

Table 11 showcases the mesh shader based pipeline's warp occupancy (per streaming multiprocessor) as a percentage of the theoretical maximum. Figure 12 visualises this data into a graph.

VERTEX PIPELINE			**Warp Occupancy (%) (Per SM)**			
Max High LOD Dist (world-space unit)			Height (world-space unit)			
	50	100	150	200	250	300
100	9.32352	14.2217	22.249	33.7759	40.8935	41.1016
150	9.17117	13.123	21.7646	33.606	41.0793	41.0716
200	9.26177	13.1279	15.6719	32.4911	41.4739	41.1821
250	9.15998	13.052	15.2234	17.8021	40.1536	41.2655
300	9.31768	13.1455	15.181	15.7011	20.0109	40.029

Table 11, Mesh shader warp occupancy (per SM) values.

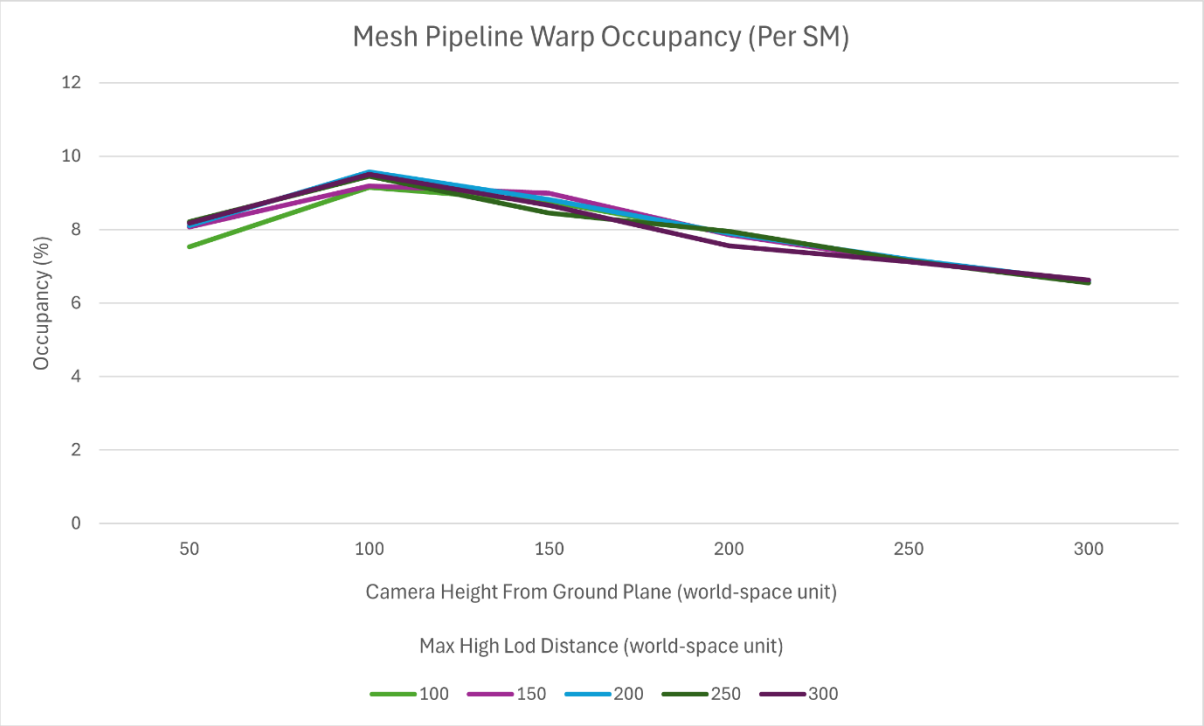

Figure 12, Mesh shader warp occupancy (per SM) values.

Chapter 5 – Discussion

This chapter will discuss and analyse the gathered data showcased within chapter 4.

5.1 Duration

When looking at the duration of the vertex shader based pipeline in table 4 / figure 5, it can be observed that all max high LOD distances appear to have near identical performance at heights of 50 and 100. This is likely due to these max high LOD distance values rendering all the grass blades within view at a high LOD level. It can be noted that typically performance appears to improve substantially when the majority of grass blades rendered switch to low LOD. This can be seen particularly in the 150, 200, and 250 values for max high LOD distance where, upon reaching the height equal to their max high LOD distance value the expected trajectory of their performance decrease seems to dip slightly. It does not decrease as much as expected, with this likely being due to a number of blades within view now being rendered as low LOD. Furthermore, upon reaching a height which surpasses the max high LOD distance value, this dip in the performance decrease becomes so large that performance actually increases. The resulting performance increase can be as high as 2E10^6 ns in the case of the 250 max high LOD test, when transitioning from a height of 250 to a height of 300. The increase in performance appears to correlate with the max high LOD distance inferring that if a measurement were taken for 300 max high LOD distance at a height of 350 world space units, this same performance gain would be observed.

Unlike the vertex shader based pipeline, the mesh shader based pipeline appears to follow a far more uniform and predictable curve, as shown in figure 9, with all max high LOD distances starting off near identical. This is likely due to all rendered grass blades being at the highest LOD, possibly with some minor variance in the furthest drawn grass blades which could explain the slight performance difference. All max high LOD distance values appear to follow a clear trend of decreasing performance as the height and blade count increases. It can be seen that performance will decrease by a smaller margin when the height is equal to the max high LOD distance however, it appears to decrease in a strictly uniform manner after surpassing this height. It can be seen that at a height of 300 all max high LOD

distance values tested (barring 300) appear to converge and continue the near identical pattern which can be seen at heights of 50 and 100. Inversely this is likely now caused by all rendered grass blades being at the lowest LOD. The max high LOD distance value of 300 appears to perform slightly worse than the other values at a height of 300 however, taking into account the pattern which can be seen with other values, it can be predicted that at a height of 350 it would achieve a near identical performance to all other tested values.

When looking at the raw data showcased in table 4 and table 8 it can be seen that the mesh shader based pipeline appears to perform noticeably better than the vertex shader based pipeline up to and including a height of 150 across all max high LOD distance values tested. At a height of 200 there is no pipeline which clearly outperforms the other, with the vertex shader based pipeline having a lower duration with a max high LOD distance of 100, 150, and by a very marginal amount 300. However, at a max high LOD distance of 200 and 250 the mesh pipeline appears to perform noticeably better. When reaching a height of 250 it can be seen that the vertex shader based pipeline begins to largely outperform the mesh shader based pipeline at all max high LOD distances excluding 250 which still maintains a slight performance lead. Once at a height of 300, the vertex shader based pipeline vastly outperforms the mesh pipeline at all max high LOD distance values tested.

Analysis of the data suggests that the mesh shader based pipeline performs best when it is able to utilise the continuous nature of its LODs as it seems to distinctly perform better at heights where there is no clear cut LOD for all grass blades to be at. This can then be seen in reverse at a height of 300 where all grass blades will be in the lowest LOD state where it performs less effectively than the vertex pipeline. Unlike the mesh shader based pipeline the vertex pipeline’s performance seems almost entirely dependent on how much grass is rendered at each LOD level, being particularly affected by high LOD grass. This is supported by the clear steps that can be seen when looking at the data where, if the max high LOD distance is equal to or greater than the camera height, the performance will be inferior than if the distance was less.

When considering the aim and research question for this project, duration is the key and primary metric for evaluation as, in a real world application such as a video game, this is the most important factor that will impact the player experience – with it correlating directly to the achievable frame rates for each pipeline. The mesh shader based pipeline seems to be viable for a close to medium distance and for relatively small patches of grass - observably up to around 240,000 blades. However, it seems for larger grass use cases such as vast fields or at further distances the traditional vertex based pipeline appears to be better suited.

5.2 L2 Cache Hit Rate

When examining the data presented in table 5 and figure 6 it can be seen that the cache hit rate for the vertex shader based pipeline never falls below 85%. It appears to generally trend upwards as more grass is rendered and additionally seems affected by high LOD grass following the same steps visible for duration in table 4 (where max high LOD distance is equal to or greater than height). One general outlier to the upwards trend is the max high LOD distance of 100, which appears to have a lower hit rate percentage than the other values tested from heights 100 to 250, this could support the theory that the hit rate % is impacted directly by the number of high LOD grass blades rendered.

The mesh shader based pipeline's L2 cache hit rate (visible in table 9 and figure 10) appears to generally follow the same upwards trend as the vertex shader based pipeline. However, the individual values are tightly clustered across the range of heights and max high LOD distances tested. It does appear that the pipeline follows the same pattern as the vertex shader based pipeline, with an increased hit rate when rendering higher LOD grass and the same distinct steps in hit rate visible - particularly at a max high LOD distance 150 and a height of 150, as well as a max high LOD distance of 200 and a height of 200. This trend does not appear to continue past a height of 200. This can be seen at a height of 250, where the max high LOD value of 250 showcases the second lowest hit rate across the 5 tested max high LOD values at this height. At a height of 300 there does not appear to be any discernible pattern for which max high LOD distance value has the highest hit rate; however, all tested max high LOD values appear to have a slightly higher

average hit rate at this height, indicating that the hit rate is likely tied to the amount of grass blades being rendered within the scene.

When comparing both pipelines' L2 cache hit rates it would appear that the vertex shader based pipeline outperforms the mesh shader based pipeline overall by a significant margin; with not a single recorded instance of the mesh shader based pipeline scoring a higher hit rate for any max high LOD distance value at any tested height. As both the vertex shader and mesh shader used for grass rendering are programmatically almost identical within their relevant grass rendering code, this discrepancy highlights the differences within the optimisation level of a standard pipeline's cache access patterns as opposed to a mesh shader based pipeline where it is far more dependent on the developer to ensure cache coherent data access patterns are being utilised.

5.3 Streaming Multiprocessor Throughput

Streaming multiprocessor throughput data for the vertex shader based pipeline can be seen in table 6 and figure 7. When analysing the data it can be seen that at a height of 50 all max high LOD distance values appear to perform almost identically however, when moving up to a height of 100 a bizarre pattern begins to emerge. It can be seen that at a height of 100 the max high LOD distance of 100 has the highest streaming multiprocessor throughput, with a value of 16% of the theoretical maximum – all other max high LOD distance values however have a far lower percentage, being between 10% and 11.5%. Moving up to a height of 150 a similar pattern can be seen; at this height the max high LOD distance value of 150 has the highest percentage at 32%. The second highest is then held by the max high LOD distance value of 100 with 23% and then much like the height of 100, all max high LOD distance values higher than the current height perform almost identically - each achieving a throughput of roughly 19%. This pattern continues all through the table with the max high LOD distance that is equal to the height value achieving the highest throughput, followed by the max high LOD distance values lower than the height and then finally followed by those higher than the height. Of course, it should be noted that the exact values for each entry increase as the height increases.

Unlike the vertex shader based pipeline, the mesh shader based pipeline does not follow this trend and instead appears to follow its own pattern. When looking at the height of 50, almost all max high LOD distance values have identical percentages - with every value greater than and including 150 achieving a streaming multiprocessor throughput of roughly 18% with a variance of only 0.32%. The max high LOD distance of 100 does not share this percentage however, achieving a slightly lower throughput at 16.2%. At a height of 100 the percentages change drastically – all recorded percentages are within 2.7% of each other between just 9.1% and 11.8%. Unlike at a height of 50, at a height of 100 the max high LOD distance value of 100 achieved the highest throughput, and the value of 300 achieved the lowest. At a height of 150 the general trend once again appears to shift, with the highest measured throughput being with a max high LOD distance of 200, which achieved a throughput of 15%; max high LOD distance values both greater than and less than 200 appear to achieve very similar throughputs at around 13% +/- 0.51%. This is the first instance of the main general trend for the mesh shader based pipelines streaming multiprocessor throughput, a max high LOD distance that is 50 world-space units greater than the height being measured appears to have the highest throughput percentage; this trend is true for heights 150, 200, and 250. At a height of 300 all measured max high LOD distance values achieve similar throughput at 11% however from the previous trend it can be inferred that if a value of 350 were measured it would achieve a slightly higher throughput percentage.

Comparing the two pipelines It is immediately clear that overall the vertex shader based pipeline generally achieves vastly higher throughput than the mesh shader based pipeline. Both pipelines begin with similar throughput however it can be seen that beyond a height of 100 the vertex pipeline achieves significantly higher percentages than the mesh shader based pipeline. The low throughput on the mesh shader based pipeline could potentially be due to the fixed thread group size used by the mesh shader, resulting in a large number of threads doing effectively no work on mesh shader instances where the grass would be a low LOD. This is an area for potential improvement in future work where, if a solution were found to this issue, a vastly higher throughput could potentially be achieved.

5.4 Warp Occupancy (Per Streaming Multiprocessor)

For the vertex shader based pipeline, looking at the data visible in table 7 and figure 8, it can be seen that at a height of 50 all max high LOD distance values have almost identical occupancy; this is likely due to all values rendering all grass blades at a high LOD at this distance, meaning the work done by all max high LOD distance values tested would be roughly equal. However, when moving to a height of 100 a slight pattern begins to emerge where the max high LOD distance of 100 has the highest occupancy at 14.2% and all other max high LOD distance values tested achieve an occupancy of around 13% at this height. A pattern of max high LOD distance values that are equal to or less than the height being tested achieving the highest occupancy then continues for all further heights, with the gap between these and the occupancy of max high LOD distances greater than the height only increasing as more grass blades are rendered. This appears to highlight a distinct correlation between the predominant LOD of grass being rendered and the occupancy percentage achieved: the more grass at a low LOD that is rendered, the higher the occupancy percentage appears to be. This can particularly be seen with the height of 250 where the max high LOD distance value of 250 and values less than this all achieve an occupancy percentage between 40% and 41% however, the value of 300 - which will be rendering significantly more grass at a high LOD - only achieves an occupancy percentage of 20%.

When looking at the mesh shader based pipeline data, presented in table 11 and figure 12, a distinctly different relationship emerges. All max high LOD values appear to have roughly equal occupancy at each height, with the largest variance being just 0.7% at a height of 50. The occupancy percentage appears to peak at roughly 9.5% at a height of 100 before dropping gradually to 6.5% at a height of 300. It appears as though the occupancy is only affected by the number of grass blades rendered however this alone does not explain the peak at a height of 100. The low occupancy of the mesh shader based pipeline appears to be indicative of poor parallelisation likely as a result of developer error. A standard pipeline will not face this issue as much of the parallelisation of a vertex shader based pipeline is handled automatically and is less developer reliant. This is a key area of improvement which could be explored in future work.

When comparing both pipelines, it is clear that the vertex shader based pipeline achieves a higher occupancy percentage at all measured heights – with the gap only increasing as more grass blades are rendered. It should be noted that low warp occupancy is not universally problematic however, in the case of the mesh shader based pipeline it appears to be a result of low instruction issue efficiency which is far less prevalent on the vertex shader based pipeline. This is an important area for further investigation in future work and could be key to improving the mesh shader based pipeline’s performance - particularly in scenes with vast grass fields.

Chapter 6 – Conclusion

6.1 Overview

This research was conducted to investigate the viability of utilising mesh shaders to improve the performance of grass rendering through the use of continuous LODs as opposed to traditional discrete LODs. To investigate this an application was developed featuring a modern geometry based grass rendering approach for both a traditional vertex shader based pipeline, and a more modern task and mesh shader based pipeline taking inspiration from previous work on grass rendering by Wohllaib E. (2022) and grass rendering with mesh shaders by Faber C. et al. (2024).
A system for culling and then organising progenerated grass clump positions by LOD was developed and allowed for discrete LODs to be rendered by the vertex shader based pipeline. Additionally, a system for continuous dynamic LODs with a varying vertex count was developed and implemented for the mesh shader based pipeline. Performance metrics for both developed grass pipelines were gathered and evaluated to investigate the viability of the utilisation of mesh shaders to enhance grass rendering performance. It was found that for scenarios with a blade count below around 240,000 and at relatively close to medium distances the mesh shader based pipeline can outperform the traditional vertex shader based pipeline by as much as 36% in select cases. Beyond these scenarios however, it was found that a traditional vertex shader based pipeline with discrete LODs appears to outperform the modern mesh shader based pipeline. This can be seen particularly in scenarios featuring very large blade counts at high distances where the continuous LODs were beyond their maximum range and had no edge over the discrete LODs which can benefit from less computation being required.

6.2 Future Research

There are several aspects of the mesh shader based pipeline which could be improved and would likely result in some performance gain, potentially leading to the pipeline outperforming the traditional vertex shader based pipeline in all scenarios. Based on the findings from the evaluated data one key area for improvement appears to be parallelisation, with the current implementation suffering from low warp occupancy (per streaming multiprocessor) which appears to be a symptom of low

instruction issue efficiency. This could likely be improved through stricter adherence to the optimal practices for mesh shading and through more efficient dispatching from the task shader.

Another area of improvement could be the use of shared memory; this would remove the unnecessary calculations the current implementation has for each mesh shader thread. Certain parameters such as the blades Bezier control points could be calculated a single time for each blade then stored in shared memory to be accessed by each vertex of that blade - reducing redundant calculations and likely leading to a significant increase in performance.

While this research only investigated up to 871,264 blades at a distance of 300 world-space units, it may be valuable to extend the research to include the impact of higher blade counts at further distances. If carried out, it would be important to implement even lower LOD levels, possibly down to billboarded grass similar to that discussed in chapter 2.7 – much like the grass implemented by Pelzer K. (2004). This type of study could provide a more thorough investigation into the viability of a mesh shader based grass system to be utilised universally within an application such as a large scale open world game.

References

Patidar S. Bhattacharjee S. Singh J M. Narayanan P J. (2007) *Exploiting the Shader Model 4.0 Architecture* [Online] [Accessed 29 April 2026] Available at: https://cdn.iiit.ac.in/cdn/cvit.iiit.ac.in/images/Thesis/MS/skpMS2009/papers/gfxSM4.pdf

Paszkowski R. (2024) *Mesh Shaders – The Future of Rendering* [Online] [Accessed 14 April 2026] Available at: https://www.youtube.com/watch?v=3EMdMD1PsgY&t

Wohllaib E. (2022) *Procedural Grass in 'Ghost of Tsushima'* [Online] [Accessed 14 April 2026] Available at: https://www.youtube.com/watch?v=Ibe1JBF5i5Y

Overvoorde A. (2016) *Vulkan Tutorial* [Online] [Accessed 3 April 2026] Available at: https://vulkan-tutorial.com/Introduction

Jansson E. (2024) *GPU driven Rendering with Mesh Shaders in Alan Wake 2* [Online] [Accessed 30 April 2026] Available at: https://www.youtube.com/watch?v=EtX7WnFhxtQ

Faber C. Kuth B. Meyer Q. Oberberger M. (2024) *Procedural grass rendering* [Online] [Accessed 3 April 2026] Available at: https://gpuopen.com/learn/mesh_shaders/mesh_shaders-procedural_grass_rendering/

Maughan C. Wloka M. (2001) *Vertex Shader Introduction* [Online] [Accessed 11 March 2026] Available at: https://developer.download.nvidia.com/assets/gamedev/docs/NVidiaVertexShadersIntro.pdf

Ebner M. Reinhardt M. Albert J. (2005) *Evolution of Vertex and Pixel Shaders* [Online] [Accessed 11 March 2026] Available at: https://stubber.math-inf.uni-greifswald.de/~ebner/resources/uniWu/evoShader.pdf

Vulkan Documentation [Online] [Accessed 19 March 2026] Available at: https://docs.vulkan.org/spec/latest/index.html

Wikipedia. (2025) *Geometry Instancing* [Online] [Accessed 20 March 2026] Available at: https://en.wikipedia.org/wiki/Geometry_instancing

Carucci F. LionHead Studios. (2005) *Inside Geometry Instancing* [Online] [Accessed 20 March 2026] Available at: https://developer.nvidia.com/gpugems/gpugems2/part-i-geometric-complexity/chapter-3-inside-geometry-instancing

Khronos Group. (2017) *OpenGL 4.6 Reference Pages* [Online] [Accessed 20 March 2026] Available at: https://registry.khronos.org/OpenGL-Refpages/gl4/

Sunar M S. Zin A M. Sembok T M T. (2008) *Improved View Frustum Culling Technique for Real-Time Virtual Heritage Application* [Online] [Accessed 20 March 2026] Available at: https://d1wqtxts1xzle7.cloudfront.net/54307836/Improved_View_Frustum_Culling_Technique_20170831-2640-1x0dmor-libre.pdf?1504242994=&response-content-disposition=inline%3B+filename%3DImproved_View_Frustum_Culling_Technique.pdf&Expires=1774019284&Signature=NOzuYcScibJK09ESMxIZ9L27TT5Eola687NzslSbnzU0FeuIdrreyBKcoTEF~Aa5MQEZoVQSw4QdMpio~4I3gDME4ukkhYTfqNA79LCnvaotmJaIULXqZprWjwYxS2tV4itbOCi15fD6cKz1ZnYG2VDQnftgTmXZ~qTkZrkjuhhkMpI-pSPe1Eit2IKXuMsQDatk~RFLJOgq5QGk~i6A4u5kVceXaBa0H0qx0whlUBWMmqkpC-gd2Mbosoyc7eQliMWlCD36ufHCAfW8pZWwD3qzykMGNY0UloSfTQXuD~PUi9DgWF916wnl3FEbHcUTjdjTPwfF05EYGZ4~QasMPA__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA

Oberberger M. Kuth B. Meyer Q. (2023) *From vertex shader to mesh shader* [Online] [Accessed 21 March 2026] Available at: https://gpuopen.com/learn/mesh_shaders/mesh_shaders-from_vertex_shader_to_mesh_shader/

Kubisch C. (2018) *Introduction to Turing Mesh Shaders* [Online] [Accessed 22 March 2026] Available at: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/

Hargreaves S. (And many others) (2019) *Mesh Shader – DirectX Specs* [Online] [Accessed 22 March 2026] Available at: https://microsoft.github.io/DirectX-Specs/d3d/MeshShader.html#motivation-for-adding-mesh-shader

Linebender Authors (2023) *Mesh Shaders* [Online] [Accessed 22 March 2026] Available at: https://linebender.org/wiki/gpu/mesh-shaders/

Pelzer K. (2004) *Rendering Countless Blades of Waving Grass* [Online] [Accessed 22 March 2026] Available at: https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-7-rendering-countless-blades-waving-grass

Jahrmann K. Wimmer M. (2017) *Responsive Real-Time Grass Rendering for General 3D Scenes* [Online] [Accessed 23 March 2026] Available at: https://www.cg.tuwien.ac.at/research/publications/2017/JAHRMANN-2017-RRTG/JAHRMANN-2017-RRTG-draft.pdf

Beeson C. et al. (2004) *GPU Gems* [Online] [Accessed 23 March 2026] Available at: https://developer.nvidia.com/gpugems/gpugems/