Master of Science in Engineering: Game and Software Engineering
June 2019

Skeletal Animation Optimization Using Mesh Shaders

Peyman Torabi

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Engineering: Game and Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Peyman Torabi
E-mail: petb14@student.bth.se
peyman.torabi@pahlavan.se

External advisor:
Emil Persson, Senior Graphics Engineer
Epic Games, Inc

University advisor:
Adjunct Professor Stefan Petersson
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. In this thesis a novel method of skinning a mesh utilizing Nvidia's Turing Mesh Shader pipeline is presented. Skinning a mesh is often performed with a Vertex Shader or a Compute Shader. By leveraging the strengths of the new pipeline it may be possible to further optimize the skinning process and increase performance, especially for more complex meshes.
Objectives. The aim is to determine if the novel method is a suitable replacement for existing skinning implementations. The key metrics being studied is the total GPU frame time of the novel implementation in relation to the rest, and its total memory usage.
Methods. Beyond the pre-existing implementations such as Vertex Shader skinning and Compute Shader skinning, two new methods using Mesh Shaders are implemented. The first implementation being a *naive* method that simply divides the mesh into meshlets and skins each meshlet in isolation. The proposed novel *common influences* method instead takes the skinning data, such as the joint influences of each vertex, into account when generating meshlets. The intention is to produce meshlets where all vertices are influenced by the same joints, allowing for information to be moved from a per vertex basis to a per meshlet basis. Allowing for fewer fetches to occur in the shader at run-time and potentially better performance.
Results. The results indicate that utilizing Mesh Shaders results in approximately identical performance compared to Vertex Shader skinning, (which was observed to be the fastest of the previous implementations) with the novel implementation being marginally slower due to the increased number of meshlets generated. Mesh Shading has the potential to be faster if optimizations unique to the new shaders are employed. Despite producing more meshlets, the novel implementation is not significantly slower and is faster at processing individual meshlets compared to the naive approach. The novel Common Influences implementation spends between 15-22% less time processing each meshlet at run-time compared to the naive solution.
Conclusions. Ultimately the unique capabilities of Mesh Shaders allow for potential performance increases to be had. The proposed novel Common Influences method shows promise due to it being faster on a per meshlet basis, but more work must be done in order to reduce the number of meshlets generated. The Mesh Shading pipeline is as of writing very new and there is a lot of potential for future work to further enhance and optimize the work presented in this thesis. More work must be done in order to make the meshlet generation more efficient so that the run-time workload is reduced as much as possible.

Keywords: Turing Mesh Shaders, Skeletal Animation, Skinning.

Sammanfattning

Bakgrund. I denna avhandling presenteras en ny metod för att deformera en modell med hjälp av den nya Mesh Shader funktionaliteten som är tillgänglig i Nvidias nya Turing arkitektur. Deformering av modeller utförs just nu oftast med så kallade Vertex eller Compute Shaders. Genom att nyttja styrkan hos den nya arkitekturen så kan det vara möjligt att ytterligare optimera deformeringsprocessen och på så sätt öka prestandan. Speciellt i samband där mer komplexa modeller används.
Syfte. Syftet är att avgöra om den nya metoden är en lämplig ersättning av de nuvarande implementationerna. De viktigaste aspekterna som studeras är den totala GPU-exekveringstiden per bild som renderas av den nya metoden i förhållande till resterande, samt dess totala minnesanvändning.
Metod. Utöver de befintliga implementeringarna, såsom Vertex Shader deformering och Compute Shader deformering, implementeras två nya metoder som använder Mesh Shaders. Den första implementeringen är en *naiv* metod som helt enkelt delar modellen i mindre delar, så kallade meshlets och deformerar varje meshlet i isolering. Den föreslagna nya *common influences*-metoden tar i stället hänsyn till deformeringsdatan som tillhör modellen, såsom de gemensamma inverkningarna av varje vertex, vid generering av meshlets. Avsikten är att producera meshlets där alla vertriser påverkas av samma leder i modellens skelett, vilket gör det möjligt att flytta informationen från en per vertris basis till en per meshlet basis. Detta tillåter att färre hämtningar sker på grafikkortet vid körning och vilket kan potentiellt ge bättre prestanda.
Resultat. Resultaten indikerar att utnyttjandet av Mesh Shaders resulterar i ungefär samma prestanda jämfört med Vertex Shader deformering, (som observerades vara den snabbaste av de existerande implementationerna) samt att den orginella implementationen är marginellt långsammare på grund av ett högre antal meshlets genereras. Mesh Shading har potential till att bli snabbare om optimeringar som är unika till den nya arkitekturen används. Trots att man producerar fler meshlets, är den nya metoden inte markant långsammare och är snabbare med att bearbeta meshlets individuellt jämfört med den naiva implementationen. Den orginella implementationen spenderar mellan 15-22% mindre tid per meshlet vid körtid jämfört med den naiva lösningen.
Slutsatser. I slutändan så erbjuder Mesh Shaders unika nya möjligheter till optimeringar som kan leda till potentiellt bättre prestanda.

Den föreslagna nya Common Influences-metoden är lovande på grund av att den är snabbare per meshlet, men mer arbete måste utföras för att minska antalet genererade meshlets. Mash Shaders och Turing arkitekturen är vid skrivande stund fortfarande väldigt nya och det finns mycket potential för framtida arbeten att yterrligare förbättra och optimera det arbete som presenteras i denna avhandling. Mer arbete

måste utföras för att göra meshletgenereringen effektivare så att arbetet som måste utföras under körtid minskas så mycket som möjligt.

Nyckelord: Turing Mesh Shaders, Skeletal Animation, Skinning.

Acknowledgments

I would like to offer my special thanks to Emil Persson who has been my supervisor at Epic Games for the duration of my thesis work. Without his advice and help this thesis would not have been possible. I also wish to give a special thanks to Stefan Petersson who has not only been very helpful during the writing of this thesis, but also over the course of the years that I have studied at BTH. I would also like to thank Yuriy O'Donnell for the additional help and input that he has given. Lastly I wish to thank all of Epic Games for giving me the opportunity to do my thesis with them.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

A key component of three-dimensional computer graphics is animation, similar to how traditional two-dimensional animation allows for objects in a scene to move and bring life to an otherwise static scene. Three-dimensional animation operates on the same principles by creating key poses over several frames in order to create an illusion of movement over time. One ubiquitous method of animation in computer graphics is *skeletal animation*, which allows for characters to animate based on transformations provided by an underlying skeleton. The key frames driving the mesh are often created in an external tool such as Autodesk Maya. An example of this can be seen in figure 1.1, where a character which has a skeleton is being deformed at run-time based on transformations applied to its skeleton. In order to achieve skeletal animation, a process known as *skinning* [10] is performed on either the *central processing unit* (CPU) or the *graphics processing unit* (GPU). The individual vertices of the mesh are transformed based on which joints in the skeleton are influencing it. An example visualization of the joint weights can be seen in figure 1.2.

While CPU skinning is possible, it is common practice for real-time applications to perform the skinning process on the GPU at run-time. Typically the process occurs in the *vertex shader* pipeline stage [3, p. 114] where vertices are traditionally transformed into their final positions before being processed further in the subsequent stages.

It is important to note that while the traditional vertex shader skinning solution is ubiquitous, there is another method that is also common in modern game engines[1] called *Compute shader skinning* [13]. Compute shaders can be used to skin the character, and once the skinning process is completed the result can then be used in any and all pipelines that need the character. For example a graphics pass which then renders the mesh as usual, or a shadow pass which receives the mesh already skinned. This means that the shadow pass does not need to skin the mesh again in order to use it. A fourth option is the one proposed in this report, the new *Turing mesh shader pipeline*, which is very similar to a compute shader pipeline but has access to fragment output as well [8], which means it is capable of drawing to the screen. Due to the similarities with the compute shader it is paramount to have a compute implementation to compare against, resulting in a total of three different skinning variants that were compared to each other. With the latter two implementations, compute shader skinning and mesh shader skinning, being the most interesting ones.

[1] Emil Persson Senior Graphics Engineer Epic Games, E-mail 13th of June 2019.

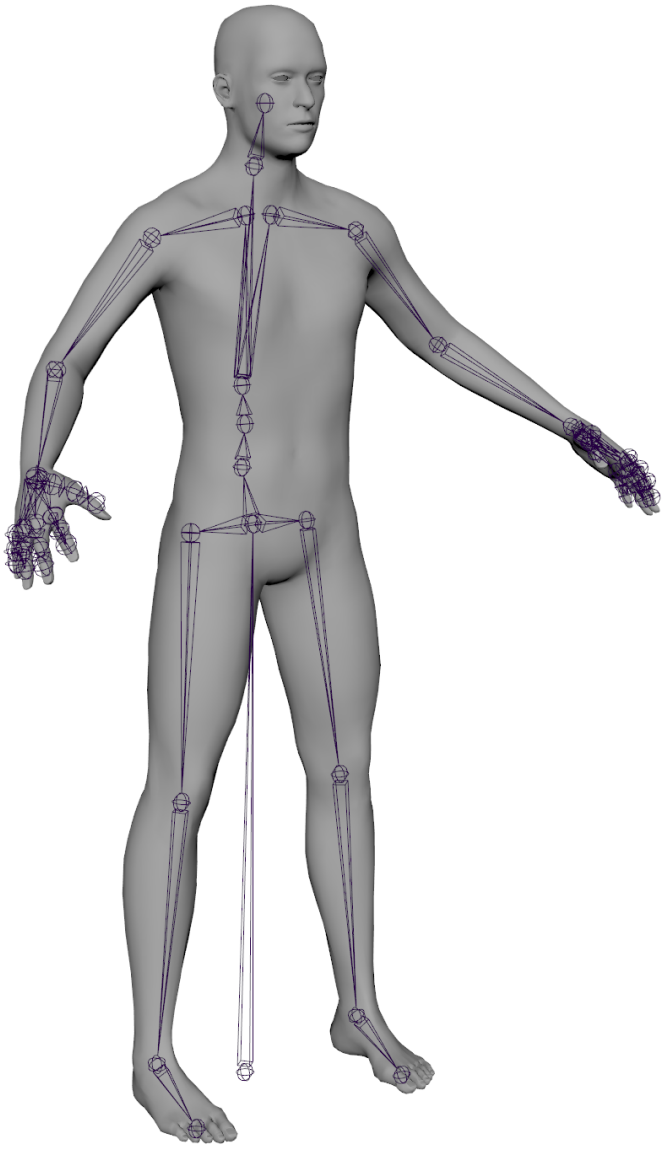

Figure 1.1: Example of a mesh and its corresponding skeleton which consists of joints. The mesh is in its so called *bind pose* where no animation has been applied.

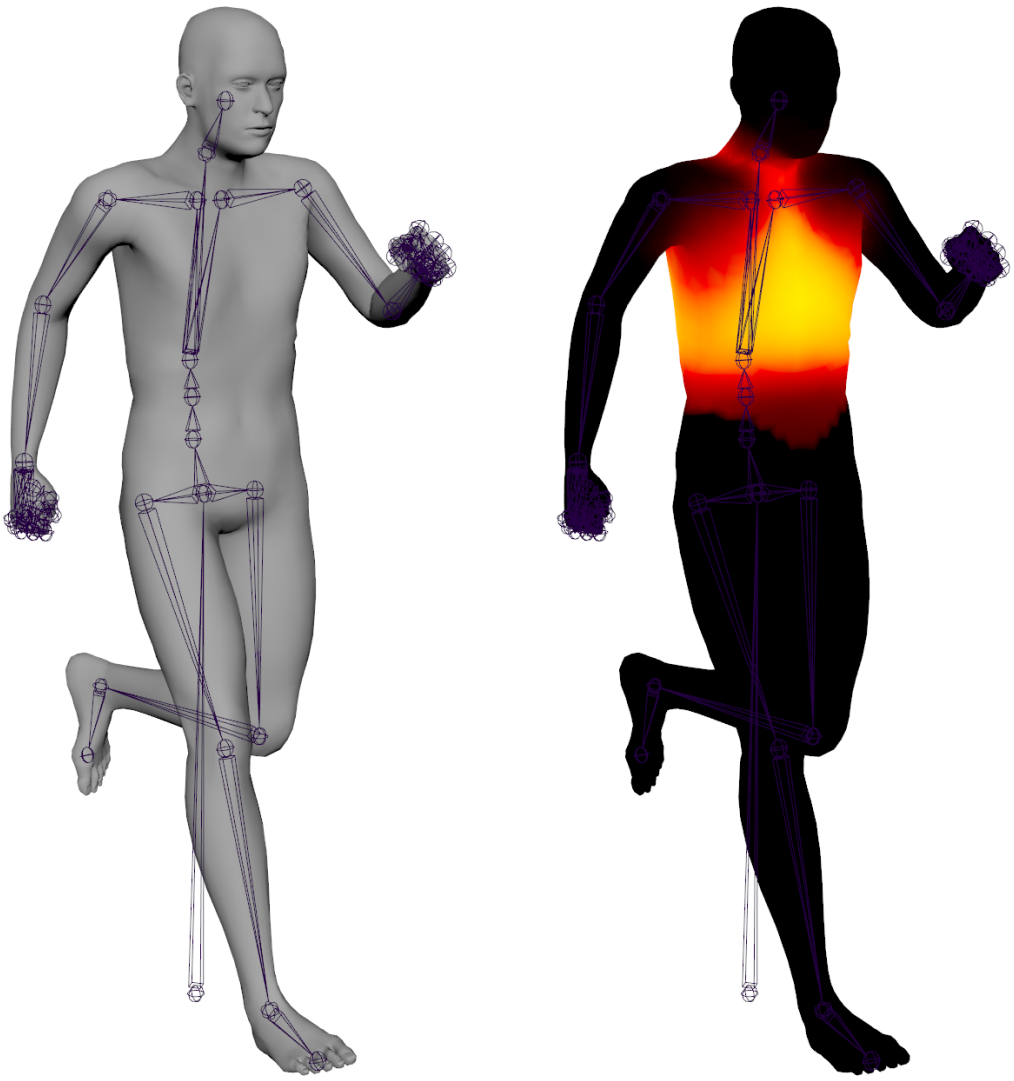

Figure 1.2: *Left:* The mesh previously shown in figure 1.1 with animation being applied to its skeleton resulting in the mesh being deformed into the given pose. *Right:* The mesh shown in figure 1.1 showcased with the joint weight influence from the chest joint in the skeleton. Black equates to no influence and yellow corresponds to full influence, meaning that the vertex will only be transformed by the joint in question as opposed to multiple joints.

1.1 Skinning with Mesh Shaders

Typically each vertex is affected by a series of weights, each corresponding to influence from a joint. Even if several vertices may be affected by the same joints the process is performed in isolation. However with the introduction of Nvidia's new programmable pipeline which was introduced with the new Turing architecture a new approach may be possible which can optimize the GPU skinning process. The new programmable shader stage known as a *mesh shader* provides a method of dividing a mesh into several meshlets which are smaller pieces that together form the original mesh. The meshlets allows the GPU to process data in larger chunks with each meshlet being processed as a piece instead of individual vertices. With this ability to divide in mind it may be worth investigating whether it is possible, and efficient, to divide a skeletal mesh into meshlets based on vertices that share common joint influences. For example, all the vertices of a character's upper left arm which are affected by the left upper arm joint of the skeleton, will be grouped together into a meshlet and skinned together, instead of being individually processed in separate pipelines. Performing GPU skinning with this new approach may allow for a larger number of skeletal meshes to be present in a scene while still maintaining good performance compared to the traditional method of skinning each vertex individually.

Skeletal animation is ubiquitous today and the ability to further optimize it is of great relevance to the industry as a whole. Several techniques exist to help improve run-time performance, such as replacing meshes with two-dimensional impostors [11, p. 481] when they are very far away, reducing the number of frames played of an animation, or hiding the mesh altogether with view frustum culling [1, p. 835]. This is why testing this new method may be of great interest as it involves investigating a novel method utilizing cutting edge hardware.

Performance is of utmost importance in rendering, even more so for real-time rendering. With the ever increasing demand for higher fidelity visuals, which is often accompanied by increasingly complex meshes, a need arises for optimizing the GPU workload as efficiently as possible. Both for present and future workloads that may occur. It is within this context that the current methods may eventually falter and a new method has to be developed. A character in a modern big budget title may typically have anything within the range of 35 to 80 thousand polygons[2], but this number may go up significantly within the coming years. A meshlet based approach can then efficiently divide these vividly complex meshes into smaller more manageable workloads based on the skeleton's weights and how they influence parts of the mesh. Thus executing the skinning process in a more efficient manner. Even if the polygon count were not to drastically increase, the new method can still improve the performance of current workloads with the added benefit of handling potential future demands as well. In other words, the intent behind developing this mesh shader based implementation is to have greater performance on a per skeletal mesh basis, and provide a higher degree of scalability.

The ultimate goal is to determine if there is a way of improving the efficiency of skinning a mesh on the GPU by use of the new Turing architecture's mesh shader pipeline.

[2]Kenneth Brown Senior Technical Artist at Epic Games, E-mail 14th of June 2019.

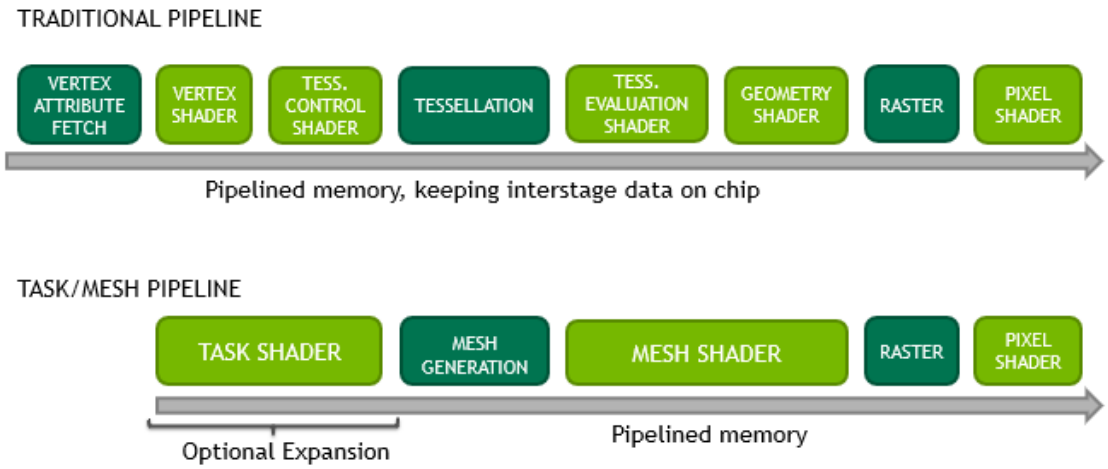

Figure 1.3: An overview of the differences between the new Mesh Shader pipeline and the old traditional Graphics Pipeline. Image created by Nvidia[8]. Accessed June 2019.

1.1.1 The Mesh Shading Pipeline

As mentioned previously the new Mesh Shader is similar to Compute Shaders. This is due to the fact that the the Mesh Shader does not operate on a per vertex basis and neither does a Compute Shader. This means that they have access to all of the relevant buffers at once, able to access any part of it at any time. This is unlike a traditional Vertex Shader pipeline where the Vertex Shader only has access to one particular vertex at a time and that one vertex's information is the only information that is accessible in that current instance. The new pipeline relies on groups of threads cooperating during execution and they are structured in an almost identical fashion in code save for a handful of Mesh Shading unique features present in the code. One may even disable the pixel output entirely and simply use the new pipeline as a compute pipeline if they wish. For a more in depth view of what the new pipeline is capable of, it is highly recommended to read through Nvidia's introduction on the subject [8].

One of the goals of the name pipeline is to replace the many steps of the graphics pipeline with a single step which is controlled by an optional Task Shader. An overview of the differences between the new and old pipeline can be seen in figure 1.3. Note how the new pipeline has significantly fewer steps in the pipeline.

1.2 Formulating Objectives & Research Questions

In order to achieve the goal of improved skinning on the GPU, several objectives must be completed. These objectives will in turn lead to the research questions being answered. The objectives are as follows:

- Create a test environment where vertex shader skinning, compute shader skinning and mesh shader skinning can be compared accurately.

- Assess the performance of each implementation in different scenarios, in particular the compute and mesh shader implementations.
- Determine the accuracy of the new mesh shader implementation.
- Provide concise data proving whether the novel mesh shader solution is an improvement over the existing methods presented.

Completing the listed objectives will allow for the research questions to be answered which are formulated as follows:

1. Will the novel mesh shader implementation to GPU skinning provide greater performance compared to Vertex and Compute shader skinning, and thus allow for more complex meshes to be skinned more efficiently?
2. Will the novel mesh shader implementation use less memory compared to the existing vertex and compute shader solutions?
3. Will the novel mesh shader implementation be more efficient in use cases where the data must be reused multiple times in different pipelines?

Chapter 2

Related Work

Many steps have been taken in order to improve the skinning process, for instance by utilizing *dual quaternion skinning* [7] which produces results that are often better than just linearly blending between key frames at the cost of memory usage and increased frame-time. Other methods that improve skinning exist as well, such as the one proposed by [9] which further improves upon the dual quaternion method. Allowing for even better skinning results by pre-computing the center of rotation for each vertex before skinning the mesh at run-time. Thus eliminating two of the biggest skinning artifacts that exist, the mesh collapsing on itself when using linear blending and the bulging artifact where a mesh instead expands outwards incorrectly when skinned with dual quaternions. Optimizing the actual skinning process itself, which is the focus of the proposed novel solution has received far less attention, however there is still noticeable progress in this regard as well to some extent. Moving the skinning process from the CPU to the GPU [2] is one of the first steps in optimizing and improving the performance of skinning. As discussed previously, skinning normally occurs on a per vertex basis and therefore ideal for parallelization on the GPU. Allowing the CPU to perform other calculations instead while the GPU skins the mesh concurrently. In order to increase the efficiency and performance even further, a great deal of work was done developing methods that allowed for compute shaders to be integrated into the rendering pipeline [14] as a whole. Supporting the graphics pipeline with additional computation that may be *reused* throughout execution which is something that was heavily utilized during this project as well. Another method of optimizing the skinning process itself, albeit in a more radical manner by proposing a method of creating automatic skinning transformations as proposed by [5]. Instead of constructing a full skeleton, a smaller subset of controls which are then used to derive the rest of the skeleton automatically. Resulting in a significant performance gain over using a full skeletal hierarchy created by hand.

The mesh shading pipeline itself is as mentioned earlier, very new but similar work has been done where the goal is to divide a mesh into smaller more manageable pieces. So called *mesh clustering* [6] which is a process where a complex mesh is divided into several smaller pieces that can be handled more efficiently. It is worth noting that this is not hardware accelerated in the way that meshlets are with the help of the mesh shading pipeline, however the aim is similar where the ultimate goal is to increase efficiency when handling complex meshes.

Chapter 3

Method

In order to be able to test the hypothesis a test environment must be created using one of the supported graphics *Application programming interfaces* (APIs), OpenGL, Vulkan or DirectX 12.

For this project Vulkan has been selected since it is appropriate to evaluate cutting edge features such as the mesh shading pipeline with a modern graphics API instead of something older such as OpenGL which does not provide the same level of explicit control. Since mesh shaders only work on the Turing architecture as detailed by [8], the only hardware requirement is to use one of Nvidia's latest 2000 series RTX graphics cards which are built on said architecture.

A basic Vulkan rendering engine will be created featuring the ability to load and animate a skeletal mesh from the FBX format[1] utilizing the asset importing library Assimp[2]. The Vulkan application will consist of three main parts, each part being one of the implementations that will be compared and evaluated. The first one being the basic vertex shader skinning implementation which will serve as a base for the remaining implementations.

3.1 Implementing the Framework

The initial and most basic implementation is the vertex shader skinning method. As explained in the introduction, each vertex has a series of weights and joints associated with it. The weights are used in conjunction with transformations from each corresponding joint that influences the vertex. All of the relevant data used to transform the vertex is sent to a graphics pipeline where the vertex shader transforms each vertex in isolation from the rest. This is the most straight forward solution and results in the mesh being transformed into a pose based on transformations from an animation. The result is then sent down the pipeline to the fragment shader where the mesh will then be rendered.

The subsequent implementations share a lot in common with the basic implementation, with the major distinction being where the skinning process occurs. When it comes to the compute shader variant a few more steps are involved. Instead of skinning the mesh directly in the vertex shader, a separate compute pipeline is executed which performs similar work to the vertex shader in the first implementation but with one difference. The compute shading pipeline is incapable of utilizing rasterization hardware directly and must rely on a regular graphics pipeline in order to display

[1] `https://en.wikipedia.org/wiki/FBX`

[2] `http://www.assimp.org/`

any output. In order to do this the compute shader outputs the transformed mesh to a buffer which the vertex shader then accesses and only performs basic perspective projection on before passing it along to the fragment shader which finally renders it to the screen as can be seen in figure 3.2. This implementation is particularly important since it is the most similar to the final mesh shader implementation. Comparing the novel implementation to this compute version is paramount since it ensures that the comparison is between similar implementations and not a one sided comparison, however it is still interesting to compare to the basic vertex shader implementation in order to see the what the potential gains will be in relation to a broader overall scope.

The final implementation is the mesh shader variant. Since as mentioned previously, the Turing architecture is very new, a lot of experimentation will occur when developing this final implementation. At its core it will be similar to the compute shader solution with the difference being that instead of having to rely on a separate graphics pipeline for visual output. The mesh shader pipeline may directly output to a fragment shader which can then render our output to the screen. The initial naive implementation will operate similarly to the compute shader during the skinning process itself. However the compute shader still operates on a per vertex basis whilst the mesh shader will operate on a per meshlet basis. A lot of work will be spent generating the meshlets in an as efficient manner as possible. The initial naive variant will simply divide the mesh into meshlets based on triangle count, but the ultimate goal is to create meshlets based on the joint weight influences, producing meshlets where the vertices all share common influences.

3.1.1 Divide the Mesh

As described by Nvidia [8], the maximum size of a meshlet is 64 vertices which can at most form 126 triangles. The meshlets can be generated at run-time with the use of the optional *task shader*. A shader dedicated to producing work that is then consumed by the mesh shaders. It is also possible to generate them prior to run-time when the application initially loads. This is the optimal choice when the data is static, as recommended by Nvidia. In the case of skeletal animation the mesh data remains unchanged during run-time and thus it is unnecessary to attempt to generate meshlets live.

Meshlets consist of a list of vertices and an index buffer that defines how the vertices will be used when creating triangles. This means that it is up to the designer to determine how the meshlets are created, meaning that there is no default configuration that defines a meshlet. However the most naive solution is to simply iterate over the list of vertices in a given mesh and count up to a desired vertex count (maximum of 64) before creating another meshlet containing the subsequent amount of vertices. This process is repeated until all vertices are a part of a meshlet. The result is a mesh that has been divided into meshlets in a very rudimentary manner and is then capable of being rendered by a mesh shader. While basic, this implementation should be sufficient for static meshes that have no form of skinning or animation applied to them. This implementation will serve as the basis for the proposed meshlet division where the meshlets will only contain vertices that are influenced by the same joints.

While the naive implementation is functional and allows for meshlets to be used

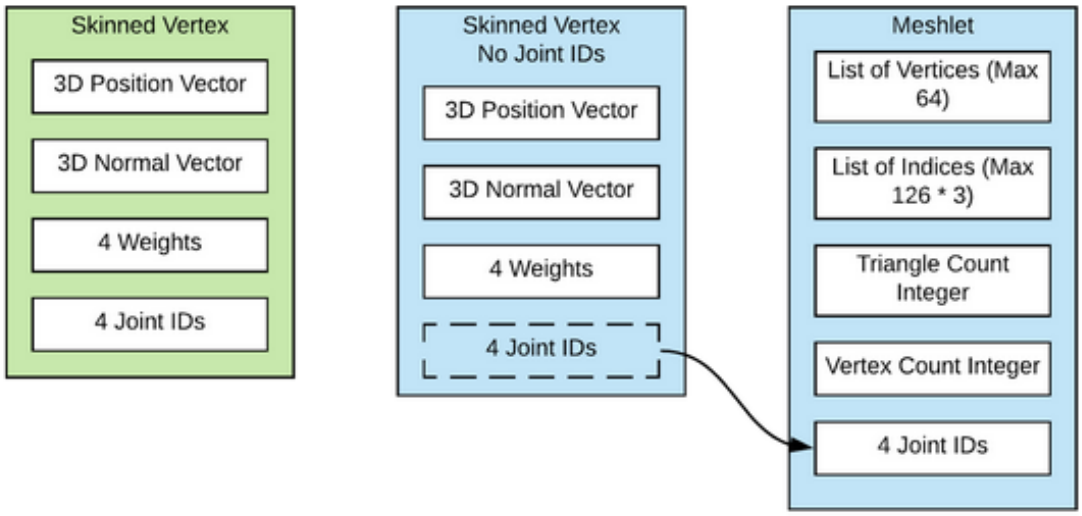

Figure 3.1: *Green:* What the vertex of a mesh may look like in code prior to implementing the proposed novel meshlet division. *Blue:* A slimmed down vertex paired with a meshlet. Note how the vertex is now slimmed down, containing less data. All vertices in an optimal meshlet share the same Joint IDs and thus this information has been placed in the meshlet that contains all of the relevant vertices. (Up to 64 vertices)

in a decent manner, further steps can be taken to improve the skinning process. Implementing a smarter meshlet creation process will allow for potential performance gains and reduced memory usage. Instead of iterating over the list and creating meshlets with triangles in the order that they are encountered, triangles will be grouped based on common influences. Triangles whose three vertices are affected by the same joints are placed together in their own group. From these groups optimal meshlets are then created where the entire meshlet consists of triangles that have the joints affecting them in common. This way vertices no longer need to contain the information required to know which four joints affect them, these so called *joint ids* may be moved to the meshlet instead. It is worth noting that all triangles do not consist of vertices that all share the same joints influencing them. These triangles must be placed in a separate remainder group and will then form meshlets using the naive solution presented earlier. Triangles such as these are ideally significantly fewer than the triangles present in the optimal meshlets as they are generally found in areas where the mesh transitions from one significant part to another. Such areas as the vertices bridging the gap between the upper thigh and pelvis of a human character.By doing this proposed novel division, information may be moved from a per vertex basis to a per meshlet basis. An example of what it may look like in code can be seen in figure 3.1. Having a slimmed down vertex, as opposed to a "fat" one, will both reduce the memory footprint of the implementation, due to there being fewer meshlets than there are vertices. It will also allow for reduced fetching on the GPU since more information is accessed once per meshlet instead of vertex, and as explained previously a meshlet consists of many vertices.

	Vertices	Meshlets	Vertices per meshlet[5]	Largest meshlet
Original mesh	54600	854	64	32 Triangles
Reduced mesh	14791	563	64	86 Triangles

Table 3.1: A comparison of a mesh before and after vertex reduction. Note the reduced number of meshlets required for the same mesh due to increased vertex reuse. The meshlets were created with the naive method described in 3.1.1.

3.2 Optimizing Mesh Data

Meshes are generally not optimal when exported from digital content creation tools such as Maya. Each triangle in a mesh has a set of three unique vertices, even if neighboring triangles have multiple vertices that are identical they are not shared between them. This is wasteful and results in there being a significant amount of duplicate vertices that bloat the vertex buffer. Before generating any meshlets it is important to increase vertex reuse in such a way that meshlets may reuse their vertices to create as many triangles as possible. The ASSIMP library used to import FBX meshes provides the ability to remove duplicate vertices when importing a mesh. A sample mesh was imported and evaluated before and after vertex reduction, the results of which can be observed in table 3.1. Vertex reduction allows for significantly fewer vertices to be present in the mesh while still providing identical visual output once rendered. As shown in the table the number of meshlets is reduced significantly due to the fact that each meshlet may contain a larger number of triangles despite having the same amount of vertices. The vertex reuse is possible thanks to *indexed rendering*[3]. A so called index buffer is used which contains indices corresponding to vertices in the now optimized vertex buffer, each sequence of three indices in the index buffer describe a triangle and the vertices it consists of. Meshlets contain their own special index buffers which describe the vertices that they contain. As described in section 3.1.1 a significant amount of time will be dedicated to optimizing the meshlets and the triangles that they consist of and thus their index buffers. In order to ensure that the methods that do not rely on meshlets are optimized as much as possible, another step will be taken to ensure this. By utilizing *vertex cache optimization*[4] it is possible to further enhance the index buffers of the other implementations. The Mesh optimizer library created by Arseny Kapoulkine[4] provides an easy to use implementation of the vertex cache optimization algorithm.

3.3 Creating Test Scenarios

The intention is to produce a method that has greater performance, so the primary metric of performance will be the frame time of the entire process as a whole (the processing and skinning of vertices to the final visual output on screen), but also the

[3] `https://vulkan-tutorial.com/Vertex_buffers/Index_buffer`

[4] `https://github.com/zeux/meshoptimizer`

[5] All meshlets contain the maximum amount vertices except for the very last meshlet created. Unless the number of vertices in the mesh is divisible by the max number of vertices chosen. The amount of benefit may vary depending on the mesh used.

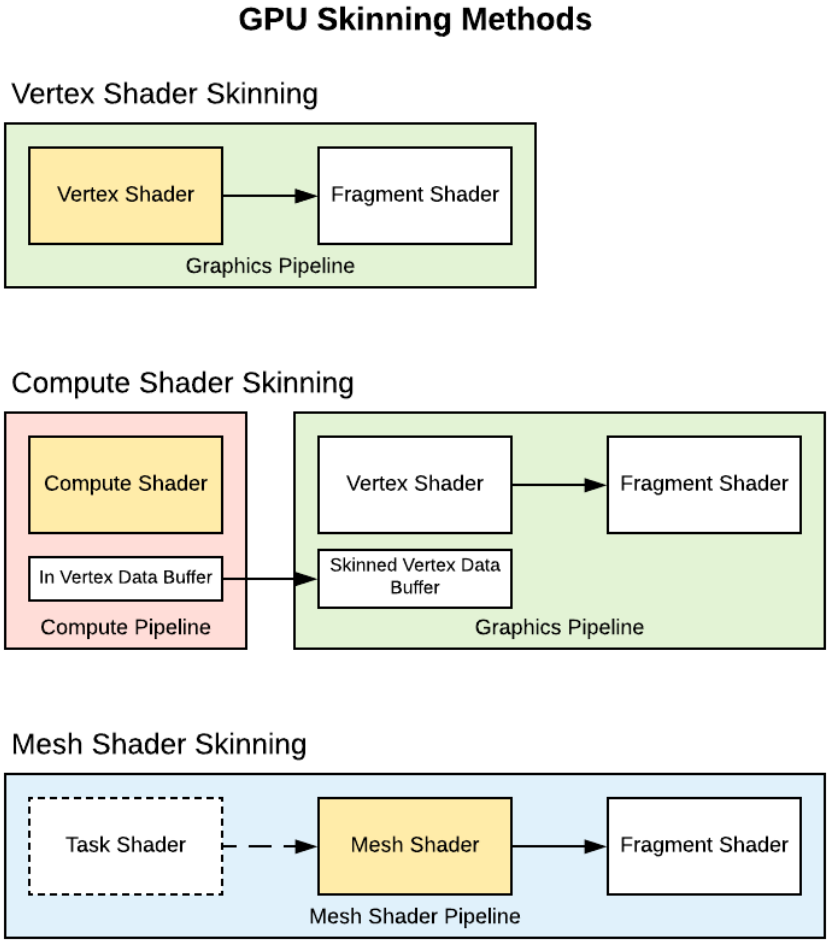

Figure 3.2: A simplified overview of the three methods of GPU skinning, including the proposed mesh shader skinning method. A dotted outline means that the shader is optional. Note that in the second compute based method, a new pipeline must be started in order to output anything to the screen. The yellow shader stages are the ones that perform the skinning of the mesh.

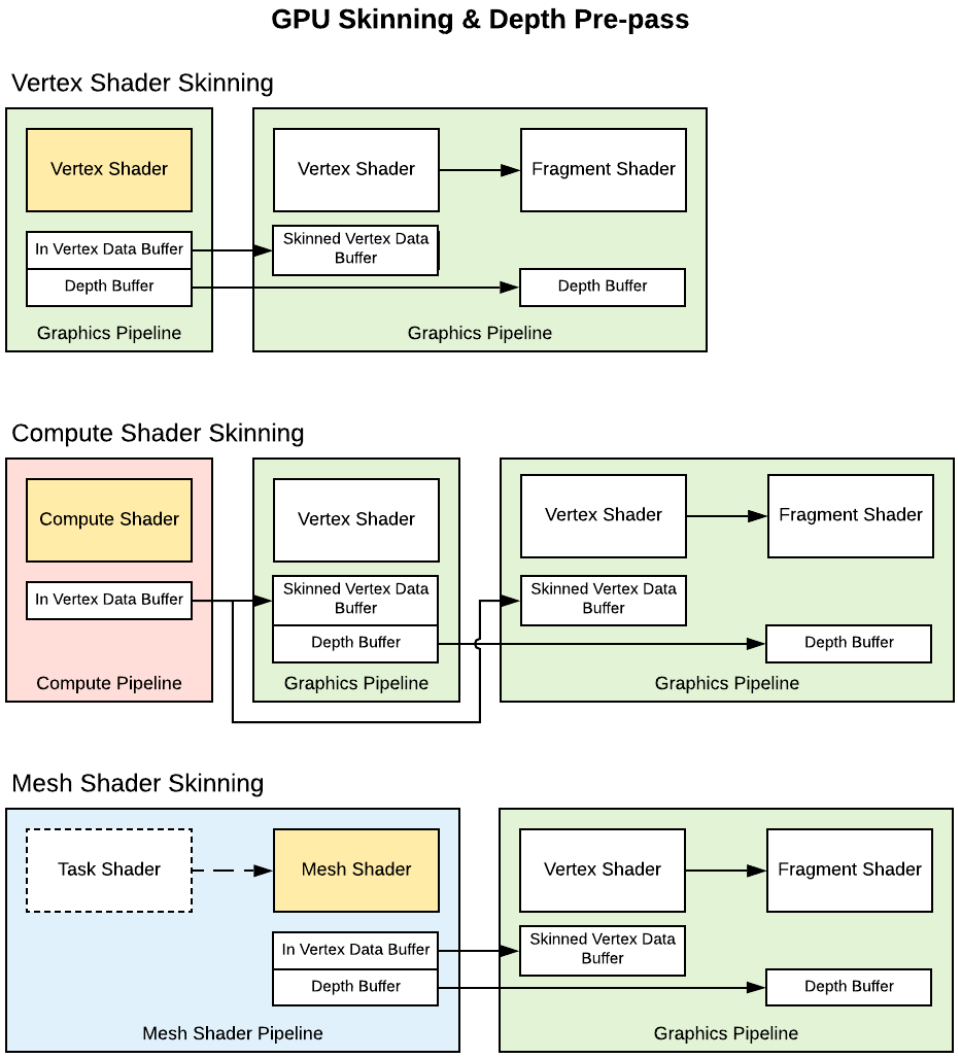

Figure 3.3: All three methods skin the mesh in a separate pipeline before sending it to an identical graphics pipeline for visual output. The secondary pass utilizes the depth buffer from the previous pipelines as well as a buffer containing the skinned mesh. Similar to how the compute shader skinning implementation functions in figure 3.2. Note that the yellow shader stages are the ones that perform the skinning of the mesh.

duration it takes to perform the actual skinning process in the respective shaders. The second metric of interest will be the video ram usage of each implementation. The intention is to make note of how much memory is used at run-time by each implementation and discern if there are any tangible differences between them. The memory usage measured will be limited to buffer allocations since buffers since despite command buffers and similar objects do require some memory, it is negligible by comparison.

Each of the previously described implementations will have two different variants. An altered environment will provide greater insight into the strengths and weaknesses of each implementation. As previously shown, in figure 3.2 the first environment will be one where each version operates in their expected fashion. The vertex shader skins the mesh and then sends it to the fragment shader. The compute shader outputs a buffer containing the skinned mesh to a graphics pipeline which then renders it, and finally the mesh shader performs the skinning inside its shader before sending it to a fragment shader. As mentioned earlier, data is often reused so it is important to create a scenario where this occurs. This is why in the second test scenario all implementations will output to a buffer, before having the skinned mesh rendered by an identical graphics pipeline. This secondary pass will also utilize a depth buffer generated by each implementation in a so called depth pre-pass, often called an *early fragment test* [12, p. 348]. An overview of this scenario can be seen in figure 3.3. The purpose of this test is to create a scenario where the skinned mesh data must be reused, similar to how data is often reused in real world applications. For instance when using a shadow pass or a reflection pass, which both require access to the skinned mesh data in order to produce the final rendered image, otherwise time is wasted by these secondary passes to skin the mesh again. Note how the compute shader skinning variant requires three passes total compared to the remaining implementations that utilize two. This is due to the fact that compute shaders are generic and do not contain anything specific to rasterization[6]. It is possible to use them for rasterization but it is not their intended use. The intention is to evaluate the implementations in scenarios that are as realistic as possible, therefore the early fragment test scenario was devised.

In the default test case will involve skinning a number of identical simple skeletal mesh consisting of relatively few polygons. A second mesh will also be utilized that will represent the complexity of a character that may be found in contemporary games, in the range of 35 - 80 thousand polygons. By stress testing the implementations it should become apparent if the novel method, or any of the others, are shown to be superior in certain cases. All tests will be executed over the course of 5000 frames and the GPU's frame time will be measured for each frame. The meshes are simply drawn on screen in rows and columns. Every instance of the mesh plays the same animation for the sake of simplicity though it is worth remembering that all meshes are treated as entirely unique by the application, no instancing or batching occurs.

[6] `https://www.khronos.org/opengl/wiki/Compute_Shader`

3.4 Ensuring Validity

In order to ensure that the tests being conducted are valid, several things have been taken into account. All three methods are performing skinning on the GPU but it is still important to remember that vertex shader skinning is very different from the proposed mesh shader solution which has more in common with how compute shader pipelines operate. Which is why a compute shader implementation has been added to the tests, bridging the gap between the vertex shader and mesh shader implementations, increasing fairness in the testing which helps ensure that the methods being compared aren't disparate. However, even if we were to produce a performant skinning solution utilizing mesh shaders it will not be worthwhile if the new implementation is not accurate. Accuracy in this case is being defined as the final output of the mesh shader skinning process being both visually and numerically identical to the output of the previous implementations. The goal is that the skinned mesh will look identical regardless of which process skinned it. This can be verified by analyzing the output skinned vertex data coordinates and comparing them between the implementations, accounting for floating point rounding errors. This way we can determine with certainty that the tests and output being produced are valid and usable.

3.5 Choosing a Test Environment

The primary focus lies with Nvidia's mesh shader pipeline the only non-negotiable hardware requirement is an Nvidia RTX series graphics card. Thus the system used both for development and benchmarking must feature an RTX 2000 series GPU or newer, the full specifications of said system can be seen in table 3.2.

Modern operating systems perform a multitude of tasks at once and it is expected that the entire system will not be devoted to the testing application. The CPU and GPU will periodically be diverted to perform other tasks concurrently separate from the test application. The CPU and GPU may also have their frequencies altered by both the operating system and the hardware controllers built into them. This may result in inconsistent performance and sometimes even poorer performance across the board since the system is continually attempting to optimize itself so that it does not consume too much power. In the case of the benchmarking maximum performance is desired in order to provide the most accurate and consistent results. Multiple steps will be taken to ensure this, such as setting the operating system power plan to *High performance* in Windows. A similar setting is present for the graphics card which sets the power management mode to maximum performance. Lastly in order to ensure consistent GPU performance, the frequency of the graphics card will be locked in order to prevent boosting which would cause the benchmarking to be inconsistent due to a fluctuating core clock. Nvidia provides a basic application for limiting the GPU core clock for this very purpose which is called *SetStablePowerState*.[7]

[7] `https://developer.nvidia.com/setstablepowerstateexe-%20disabling%20-gpu-boost-windows-10-getting-more-deterministic-timestamp-queries`

CPU	Intel Xeon Gold 6128 @ 3.4 GHz
GPU	Nvidia RTX 2080 Ti
System memory	96 GB
Operating System	Windows 10 Enterprise (v. 1709)
Graphics API	Vulkan 1.1.97.0

Table 3.2: The specifications of the system used during development and benchmarking.

Chapter 4

Results

The results presented in this chapter were generated from benchmarking with two meshes of differing complexity, both of which can be seen in figure 4.1. One basic mesh consisting of 27300 triangles and 53 joints, and one complex mesh that features 83330 triangles and a total of 155 joints. During each benchmark 2000 separate instances of the meshes were skinned over a period of 5000 frames. The two meshes are referred to as *simple* and *complex* respectively due to the latter mesh having a significantly higher amount of detail and a greater number of joints.

The implementations that utilize a combined skinning and depth pre-pass are prefixed with *DPP* and the novel common influences based mesh shader skinning implementations are postfixed with *CI* as opposed to the naive mesh shading implementations. For the sake of simplicity all implementations are also be abbreviated where suitable as follows.

- Vertex Shader - VS
- Compute Shader - CS
- Mesh Shader - MS

Frame time and memory usage are the two primary metrics observed and thus lower is better across all tests.

4.1 Recording Frame Time

All times presented are measured directly from the GPU through the Vulkan API's time stamp functionality. The times were all measured in nanoseconds and then converted to nanoseconds.[1] The CPU was not measured in these tests since the area of interest is not CPU optimization. The focus lies squarely on the GPU and how it performs during these tests. Keep in mind that despite using the same mesh 2000 times in a benchmark, all meshes are treated as separate entities and no form of instancing occurs. The meshes do not share any memory and are all processed individually as if they were all different meshes. The results of the benchmarking for the simple and complex meshes can be seen in figures 4.2 and 4.3 respectively.

[1] https://vulkan.lunarg.com/doc/view/1.0.33.0/linux/vkspec.chunked/ch16s05.html

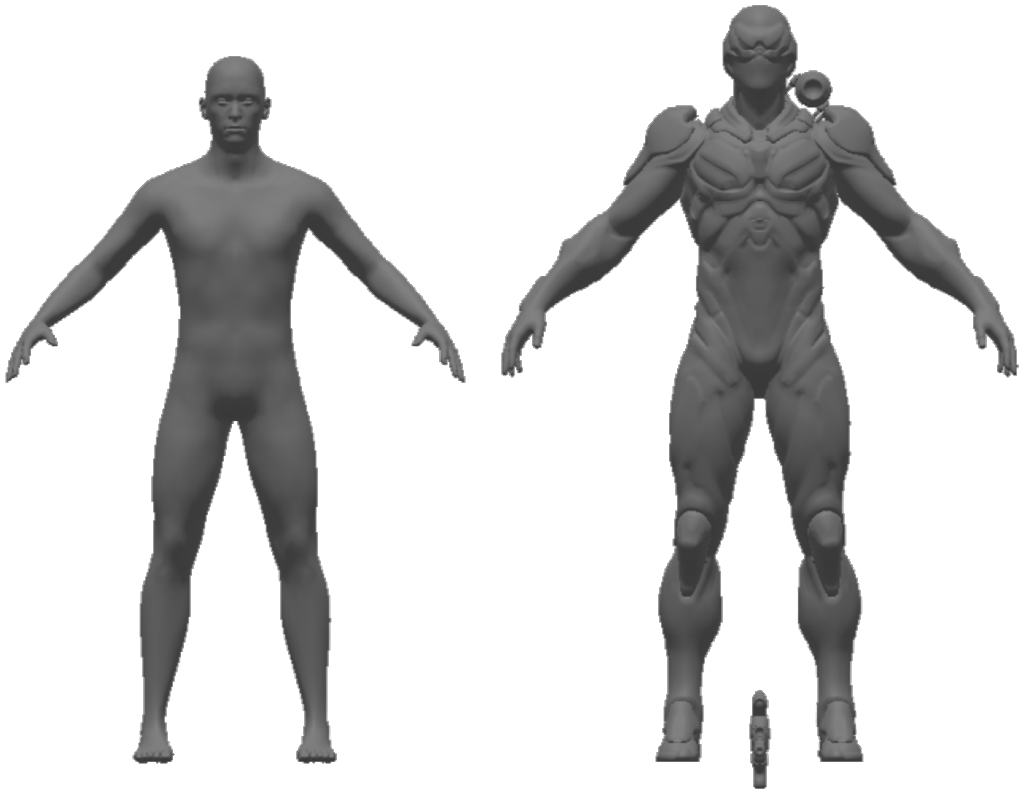

Figure 4.1: The two meshes used for the benchmarks. *Left:* A simple generic human generated with the open-source software *MakeHuman* serving as the basic case. *Right:* A complex high resolution mesh from the game *Paragon*, created and owned by Epic Games, Inc. Note that the complex character also features a weapon (which can be seen between the feet when no pose is applied) that is also taken into account during the skinning process. Also note that the picture was taken inside of Unreal Engine 4.

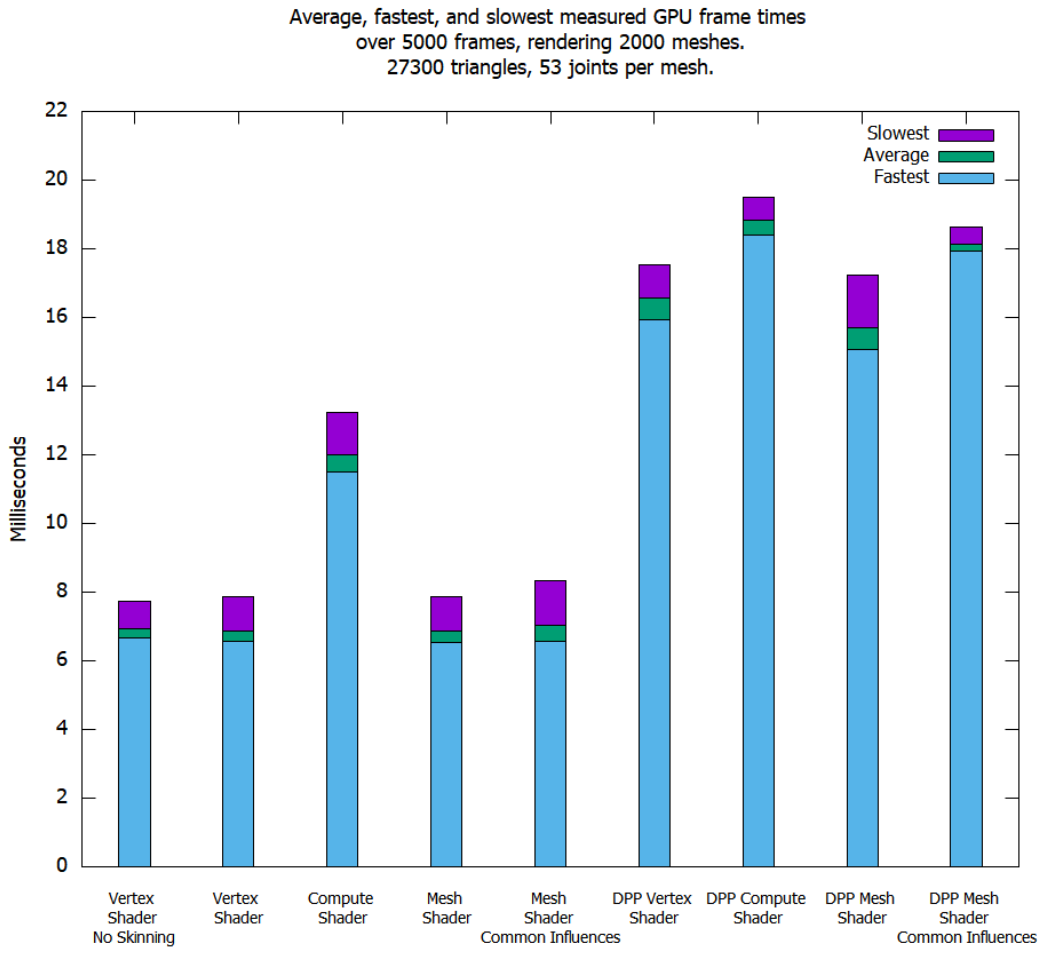

Figure 4.2: The recorded frame times of all implementations skinning the simple human mesh, including a baseline which does not skin the mesh for comparison.

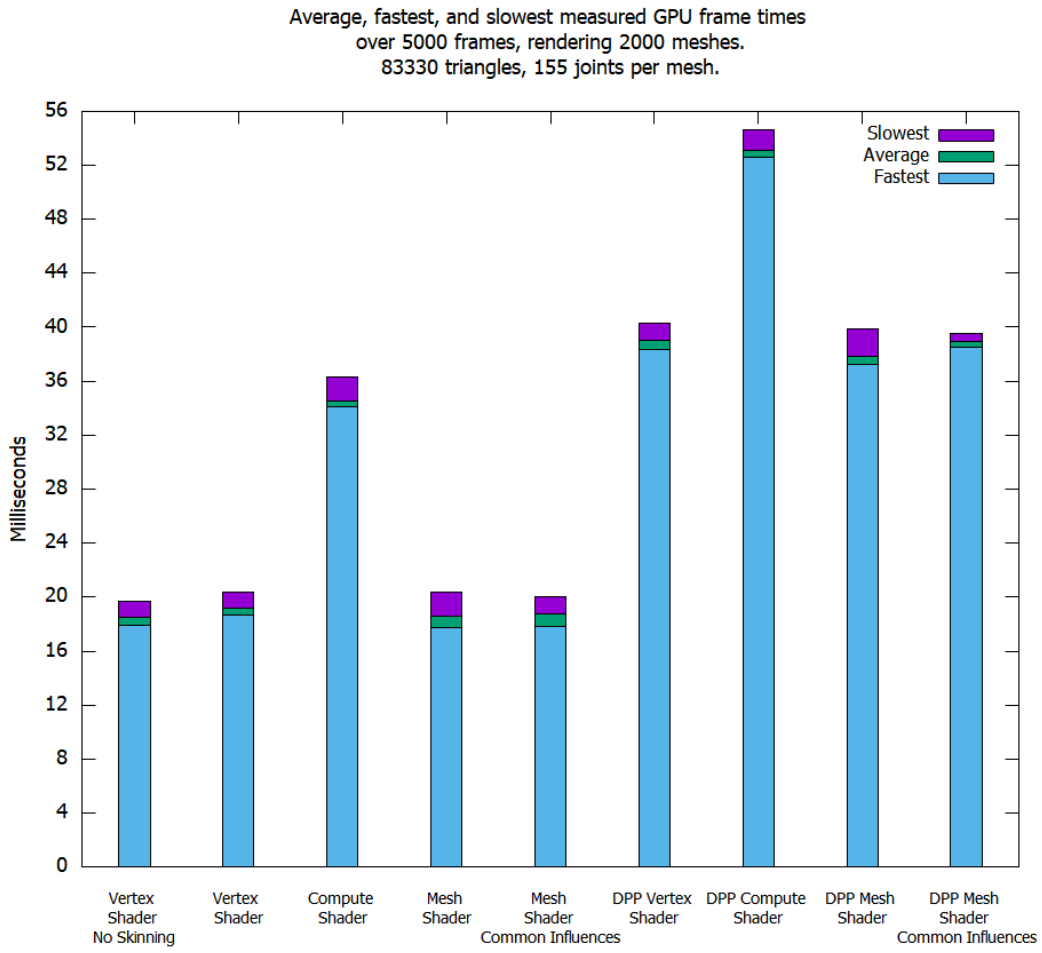

Figure 4.3: The recorded frame times of all implementations skinning the complex mesh, including a baseline which does not skin the mesh for comparison.

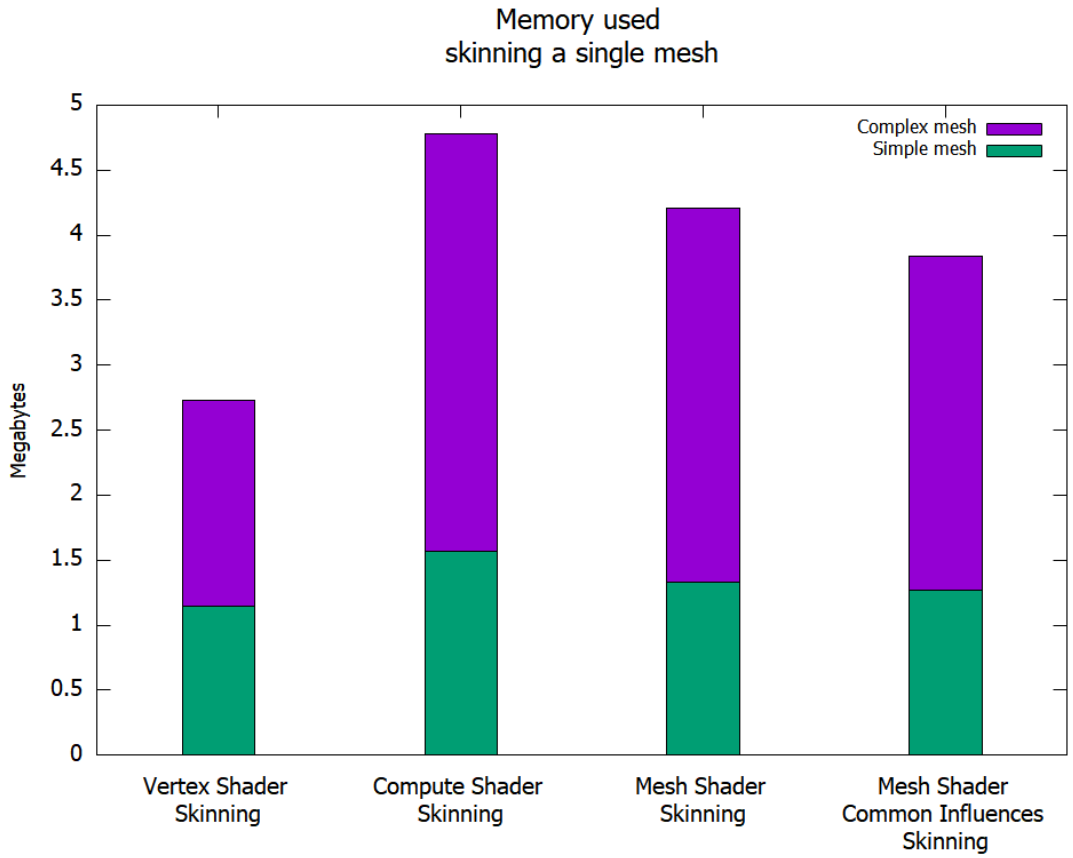

Figure 4.4: The number of megabytes used per implementation when skinning a single mesh.

4.2 Recording Memory Usage

The memory usage of each implementation was discerned from studying the memory allocations inside of Nvidia's Nsight Graphics utility software. The memory used when skinning a mesh is shown in figure 4.4. While command buffers and similar objects do require some memory on the device it is ultimately negligible, so only the memory allocated by buffers is taken into account.

4.3 Measuring Accuracy

Accuracy was determined by comparing the positions of the final skinned vertices between each implementation, ensuring that the position is identical across all variants. The simple and complex meshes were both evaluated in this manner, however due to them having thousands of vertices a very basic mesh was created for demonstration purposes only. The mesh consists of 18 vertices and 2 joints and the results of the vertices being transformed by each implementation can be seen in table 4.1. The accuracy is deemed sufficient if it is identical across all variants, taking any potential rounding errors into account.

	V0	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17
VS	-0.05	0.05	0.5407	-0.0309	0.05	0.9578	-0.031	-0.05	0.4472	-0.05	-0.05	-1	0.05	-0.05	0.8249	-0.05	0.05	-0.3873
	0	0	0.783	0.0607	-0.2873	0	0.0608	0.3075	-0.8944	0	0	0	0	0.8944	0	0	-0.8944	0
	0.05	0.4472	-0.0309	0.05	0.9578	-0.031	-0.05	0.5407	-0.05	-0.05	-1	-0.05	0.05	0.4472	0.0309	-0.05	0.4472	-0.031
	0	0.8944	0.0607	-0.2873	0	0.0608	0.3069	-0.783	0	0	0	0	0.8944	0	0.1393	-0.8944	0	0.0608
	0.4472	0.0309	0.05	0.9578	0.0309	-0.05	0.5406	0.05	-0.05	-1	0.05	0.05	0.4472	0.0309	0.05	0.4472	-0.0309	-0.05
	0.8944	0.1393	0.3075	0	0.1393	-0.2866	-0.7833	0	0	0	0	0	0	0.1393	0.5653	0	0.0607	-0.9251
CS	-0.05	0.05	0.5407	-0.0309	0.05	0.9578	-0.031	-0.05	0.4472	-0.05	-0.05	-1	0.05	-0.05	0.8249	-0.05	0.05	-0.3873
	0	0	0.783	0.0607	-0.2873	0	0.0608	0.3075	-0.8944	0	0	0	0	0.8944	0	0	-0.8944	0
	0.05	0.4472	-0.0309	0.05	0.9578	-0.031	-0.05	0.5407	-0.05	-0.05	-1	-0.05	0.05	0.4472	0.0309	-0.05	0.4472	-0.031
	0	0.8944	0.0607	-0.2873	0	0.0608	0.3069	-0.783	0	0	0	0	0.8944	0	0.1393	-0.8944	0	0.0608
	0.4472	0.0309	0.05	0.9578	0.0309	-0.05	0.5406	0.05	-0.05	-1	0.05	0.05	0.4472	0.0309	0.05	0.4472	-0.0309	-0.05
	0.8944	0.1393	0.3075	0	0.1393	-0.2866	-0.7833	0	0	0	0	0	0	0.1393	0.5653	0	0.0607	-0.9251
MS	-0.05	0.05	0.5407	-0.0309	0.05	0.9578	-0.031	-0.05	0.4472	-0.05	-0.05	-1	0.05	-0.05	0.8249	-0.05	0.05	-0.3873
	0	0	0.783	0.0607	-0.2873	0	0.0608	0.3075	-0.8944	0	0	0	0	0.8944	0	0	-0.8944	0
	0.05	0.4472	-0.0309	0.05	0.9578	-0.031	-0.05	0.5407	-0.05	-0.05	-1	-0.05	0.05	0.4472	0.0309	-0.05	0.4472	-0.031
	0	0.8944	0.0607	-0.2873	0	0.0608	0.3069	-0.783	0	0	0	0	0.8944	0	0.1393	-0.8944	0	0.0608
	0.4472	0.0309	0.05	0.9578	0.0309	-0.05	0.5406	0.05	-0.05	-1	0.05	0.05	0.4472	0.0309	0.05	0.4472	-0.0309	-0.05
	0.8944	0.1393	0.3075	0	0.1393	-0.2866	-0.7833	0	0	0	0	0	0	0.1393	0.5653	0	0.0607	-0.9251
MS CI	-0.05	0.05	0.5407	-0.0309	0.05	0.9578	-0.031	-0.05	0.4472	-0.05	-0.05	-1	0.05	-0.05	0.8249	-0.05	0.05	-0.3873
	0	0	0.783	0.0607	-0.2873	0	0.0608	0.3075	-0.8944	0	0	0	0	0.8944	0	0	-0.8944	0
	0.05	0.4472	-0.0309	0.05	0.9578	-0.031	-0.05	0.5407	-0.05	-0.05	-1	-0.05	0.05	0.4472	0.0309	-0.05	0.4472	-0.031
	0	0.8944	0.0607	-0.2873	0	0.0608	0.3069	-0.783	0	0	0	0	0.8944	0	0.1393	-0.8944	0	0.0608
	0.4472	0.0309	0.05	0.9578	0.0309	-0.05	0.5406	0.05	-0.05	-1	0.05	0.05	0.4472	0.0309	0.05	0.4472	-0.0309	-0.05
	0.8944	0.1393	0.3075	0	0.1393	-0.2866	-0.7833	0	0	0	0	0	0	0.1393	0.5653	0	0.0607	-0.9251

Table 4.1: The transformed positions of each vertex in a sample mesh (consisting of 18 vertices and 2 joints) after being processed by each implementation.

4.4 Comparing Meshlet Division Methods

In table 4.2 information regarding the performance and the meshlets generated can be viewed. The MS CI implementation's total meshlet count includes both optimal and sub-optimal meshlets. A visual representation of what the division methods actually looks like can be seen in figure 4.5. In the case of the common influences implementation, the gray parts of the mesh are still divided in a fashion similar to the naive method, with the major distinction being that the joints affecting all the vertices are shared as opposed to the red areas where the vertices may all have unique joint influences. This means that the joints affecting each vertex may be a unique set as opposed to identical across the entire meshlet. The red triangles are divided into meshlets exactly as how the naive implementation divides all triangles by simply iterating through the vertices and creating meshlet that is as large as possible. The novel common influences method's result is affected by how the mesh was weight painted in programs such as Autodesk Maya. The same mesh may produce vastly different results depending on how the weight painting was done. Two alternative results can be seen in figure 4.6 which are produced by using differing automatic skinning procedures in Maya.

4.4.1 Present Meshlet Generation Algorithms

Two algorithms were created which are used for meshlet division. One being the naive method which is an implementation of Arseny Kapoulkine's[2] meshlet division solution. The novel method is based on the naive implementation and is altered to take the skinning data into account when generating meshlets. An overview of each algorithm can be viewed in listings 4.1 and 4.2. The source code for each algorithm

[2] `https://github.com/zeux/niagara`

Simple Mesh	**MS**	**MS CI**
Average time	6.854	7.035
# of Meshlets	298	384
% # of Meshlet Delta		+∼28.859
% Avg. Time Delta		+∼2.640
Complex Mesh		
Average time	18.626	18.788
# of Meshlets	929	1079
% # of Meshlet Delta		+∼16.146
% Avg. Time Delta		+∼1

Table 4.2: A comparison of the two meshlet implementations and how they relate to each other. All times measured in milliseconds.

can also be seen in the appendix.

```
For each triangle
   Fetch triangle vertices

   if Meshlet vertex count > 64 or Meshlet triangle count >= 126
      Save Meshlet in Meshlet array

   if Vertex0 of triangle has not been encountered before
      Save Vertex0 in Meshlet's local vertex list

   if Vertex1 of triangle has not been encountered before
      Save Vertex1 in Meshlet's local vertex list

   if Vertex2 of triangle has not been encountered before
      Save Vertex2 in Meshlet's local vertex list

   Add indices of triangle vertices to Meshlet's local index list
   Increment Meshlet's triangle count
```

Listing 4.1: Naive meshlet generation pseudocode.

```
For each triangle
    Fetch triangle vertices

   if vertices have influences in common
      Save indices of vertices within a shared group
      where all vertices with common influences are stored
   else triangle does not have common influences
      Put triangle in a separate bad group

For each group of indices
   if Number of indices too few
      Put indices in the bad group in order
      to prevent optimal meshlets that consist of very
      few triangles

Create meshlets from each good group similar to naive implementation.
Create meshlets from the bad group similar to naive implementation.
```

Listing 4.2: Common influences meshlet generation pseudocode.

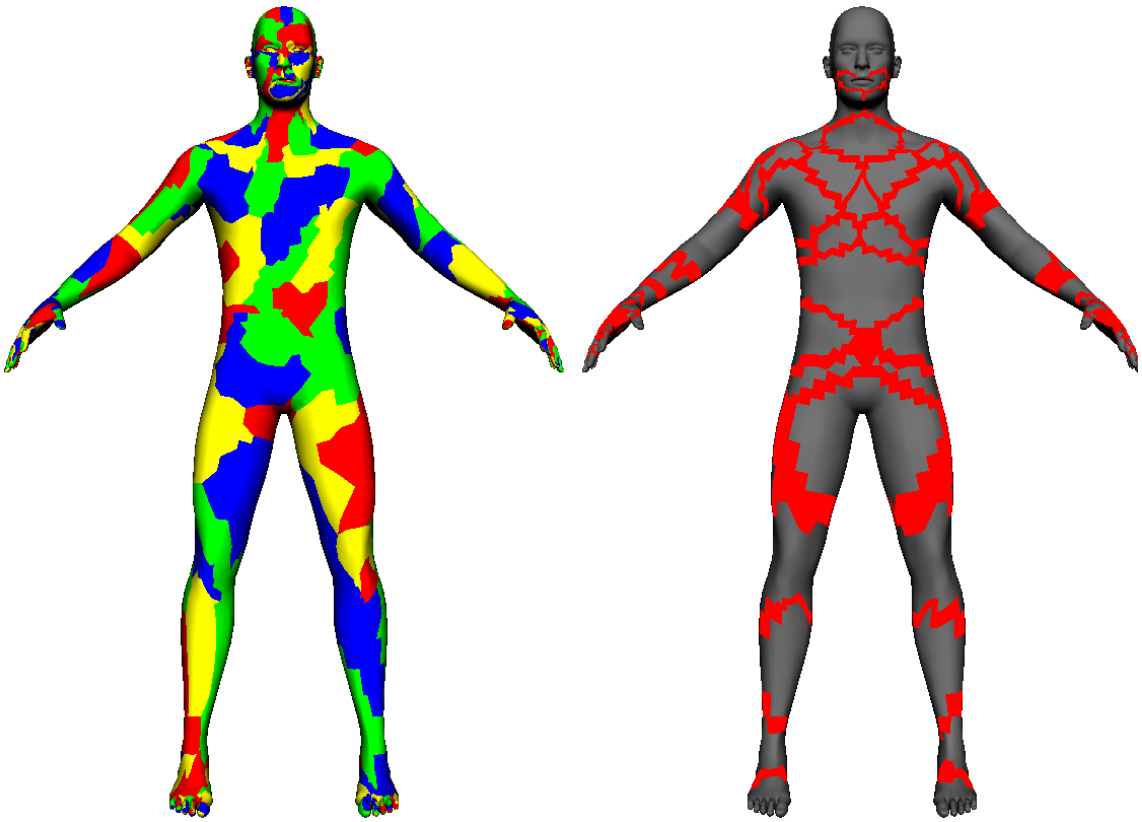

Figure 4.5: *Left:* The simple human mesh divided into meshlets using the naive approach. Note that the triangles in a meshlet do not have to form a contiguous surface. *Right:* The simple human mesh divided into meshlets using the novel common influences solution. The red areas of the mesh denote the so called sub-optimal triangles.

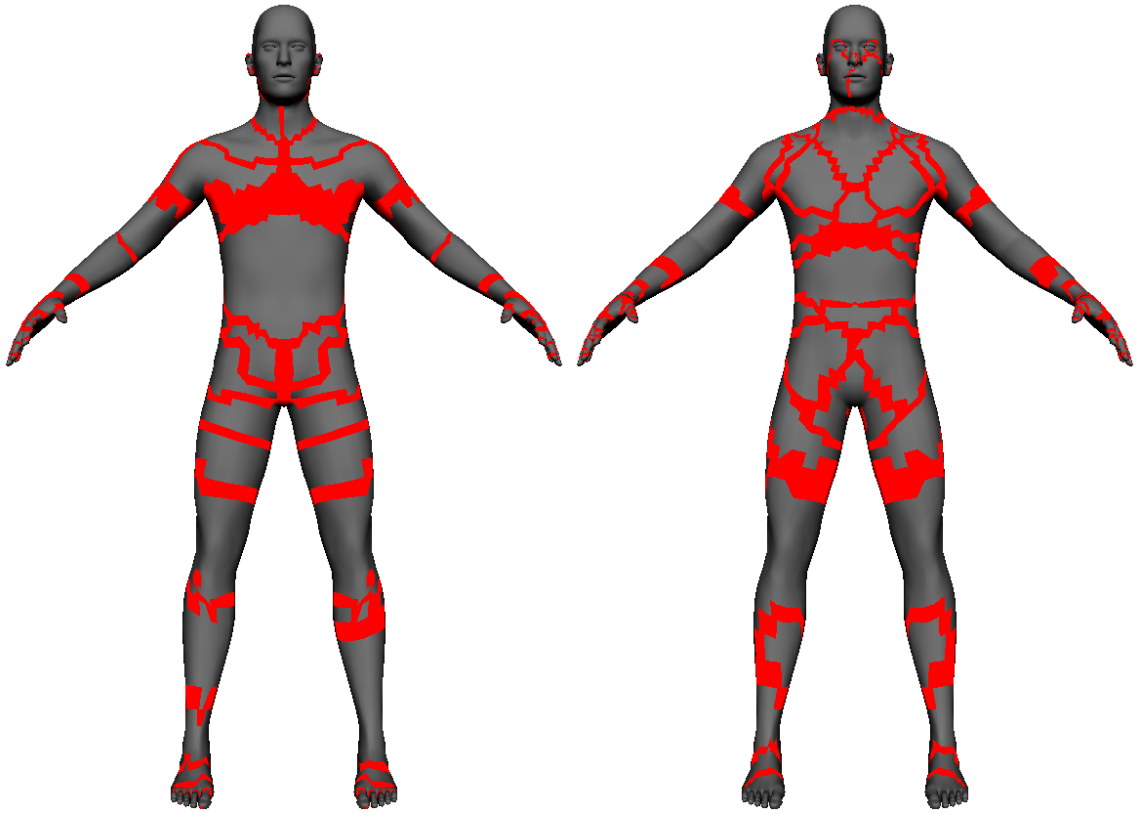

Figure 4.6: The common influences division is highly dependent on how the mesh is weight painted in an external program. Here are two alternative results each based on automatic skinning methods in Autodesk Maya.

Chapter 5

Analysis and Discussion

In chapter 4 the results of several tests and implementations were presented. In this chapter a closer look will be taken in order to understand and evaluate the results.

5.1 Evaluating Performance

As shown in figures 4.2 and 4.3, the performances of all implementations are relatively close except for the compute shader which is always slower, often to a significant degree. The main reason behind the addition of the compute shader was to provide an implementation that would be as similar as possible to mesh shading. Since Mesh Shaders differ greatly from a traditional Vertex Shader the compute shader would serve to bridge the gap. In hindsight this is expected behavior since as shown in diagram 3.2 the compute shader skinning implementation is the only implementation to feature two distinct passes. The Vertex Shader implementation will as usual directly connect to a fragment shader for output and thus only has a single pass. The Mesh Shader despite its compute-like capabilities is also able to directly output to a fragment shader as well. This lack of a secondary pass gives it a large advantage over the compute shader. Once this was established it became more pertinent to compare the Mesh Shader implementation to the classic Vertex Shader skinning solution since despite being very different, the two implementations have similar performance.

In both the simple and complex mesh test case, Mesh Shader skinning is on par with vertex shader skinning, with the Common Influences variant being a close second. Skinning the simple mesh had no perceptible difference in total time in the case of the naive implementation. The common influences variant took roughly 2.6% longer to skin the simple mesh when compared to vertex shader skinning. The complex mesh was observed to take approximately 2.8% less time with the naive implementation compared to Vertex Shader skinning. Lastly the Common Influences skinning implementation took roughly 2% less time skinning the complex mesh compared to the simple Vertex Shader.

As mentioned in section 1, Compute Shaders are often used for skinning meshes in contemporary game engines. So while Mesh Shading is not significantly faster than Vertex Shader skinning, it is faster than the Compute implementation. Since Mesh Shaders may be used as Compute shaders if one so wishes, they may be a viable replacement provided that the required hardware is available.

As seen in figure 3.3 the pre-pass variants of all implementations are significantly slower due to the fact that they consist of two render passes, except for the compute implementation which once again has an additional pass due to its inability to directly

	Simple Mesh	Complex Mesh
VS	4.4%	2.5%
CS	3.3%	2.5%
MS	5.0%	3.1%
MS CI	5.4%	3.1%

Table 5.1: An overview of the coefficient of variation for each of the implementations when skinning the simple and complex mesh. Note that the percentages are rounded up to the nearest tenth.

write to a depth buffer. Resulting in the need for a dedicated depth pass unlike the other implementations that have a combined skinning and depth pre-pass. In the case of the complex mesh the novel implementation catches up to the Vertex Shader and Naive implementations causing the difference in frame time to be negligible. Due to the limited number of meshes being used, it is difficult to discern why this only occurs with the complex mesh. As discussed earlier, the mesh being used impacts the end result of the novel implementation significantly. It may have been possible to discern the result in a more proper manner had there been a larger number of meshes being tested, as opposed to only two.

Due to the steps taken in section 3.5 the results can be regarded as much more reliable. Primarily due to the program provided by Nvidia which limits the GPU clock speed. This ensured that the results generated are as consistent as possible during testing. In order to verify this the coefficient of variation was calculated for each implementation and in table 5.1 it is made evident that the variation is very low which is good for result validity.

5.1.1 Study Vertex Cache Optimization Impact

Having benchmarks that are as fair as possible is crucial. If one implementation is significantly optimized while the others are not, the data will be biased and not provide a result that is as accurate as it could have been. As described in section 3.2, many steps were taken to ensure this. Vertex cache optimization contributes to a significant difference in performance across all implementations as can be see in figure 5.1. The reason why the vertex cache optimization plays a big part in the overall performance is due to the *Post Transform Cache*[1]. Even without the optimization this cache plays a part in the overall efficiency of the traditional graphics pipeline and the vertex cache optimization further improves the usage of the cache. Resulting in even better performance, however the Post Transform Cache can not be utilized by the Mesh Shader[2] implementations, and can only be partially used by the Compute version (during its second pass). Which can be seen in the two graphs the compute implementation is only marginally slower without the vertex cache optimization, while the Vertex Shader implementation is slowed down significantly. Another point of interest is how the Common Influences implementation is noticeably faster than all

[1] `https://www.khronos.org/opengl/wiki/Post_Transform_Cache`

[2] The cache is bound to the *vertex processing stage* found in the traditional graphics pipeline and thus can not be used anywhere else.

	MS	MS CI
Vertex Cache Optimization	298 Meshlets	384 Meshlets
No Cache Optimization	563 Meshlets	572 Meshlets

Table 5.2: The number of meshlets generated by each implementation with and without cache optimization when given the simple mesh. Fewer is better.

	MS	MS CI
Vertex Cache Optimization	929 Meshlets	1079 Meshlets
No Cache Optimization	938 Meshlets	1083 Meshlets

Table 5.3: The number of meshlets generated by each implementation with and without cache optimization when given the complex mesh. Fewer is better.

other implementations when the cache optimization is disabled, this will be discussed further in section 5.1.2.

It was initially expected that the Mesh Shader implementations would not be affected by the cache optimization whatsoever since the optimization relies on manipulating the index buffer belonging to a mesh. Since each meshlet contains its own local smaller index buffer, the large main one is not used during rendering. While it may not provide a benefit at run-time it does in fact provide a benefit during meshlet generation process, resulting in fewer meshlets depending on the mesh used. A comparison can be seen in tables 5.2 and 5.3. Note how in the case of the simple human mesh, the difference in number of meshlets is reduced significantly while in the case of the complex mesh the total meshlet count does not increase significantly. Since the number of meshlets does not change significantly for the complex mesh, the performance penalty due to disabling the vertex cache optimization is also negligible for the Mesh Shader implementations which is evident in figure 5.2. In fact, it is insignificant across all the different implementations when they render the complex mesh.

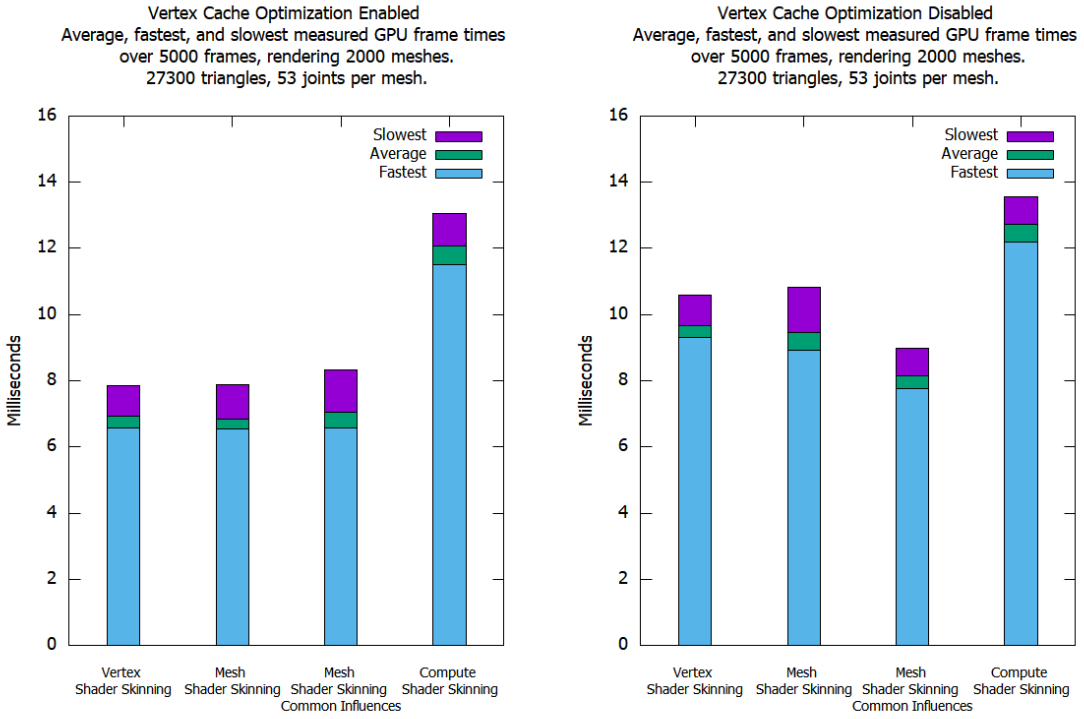

Figure 5.1: A comparison of all implementations rendering the simple mesh, with and without vertex cache optimization.

5.1.2 Compare Mesh Shading Implementations

On the right hand side of figure 4.5 the behavior of the common influences meshlet division becomes apparent. The majority of triangles are considered good, meaning that the three vertices that make up the triangle all share the same joints influencing them. The majority of the bad triangles, the triangles where the vertices are not influenced by the same joints, usually occur in transitional areas on the character. Transitional being areas between key parts of a mesh, such as where the upper and lower leg meet. In these areas the vertices are affected by different joints depending on how close they are to the next portion of the mesh. Many such borders can be seen on the chest of the character as different parts of the chest follow the clavicles and arms to some degrees while other parts of the chest are primarily affected by the chest joints.

In the tests conducted it became evident that the common influences meshlet generation algorithm produces more meshlets than the naive implementation. Because of this it is always slower, as can be seen in table 4.2. Though despite this, the slow down is not equal (percentage wise) to the increased number of meshlets. For instance in the case of the simple mesh the common influences method produces approximately 28% more meshlets but it is only around 2.6% slower. Indicating that the proposed optimization in the shader is valid and contributes to increased performance. By moving the list of joints affecting the vertices (for the optimal meshlets) from per vertex to a per meshlet basis the number of times the information has to

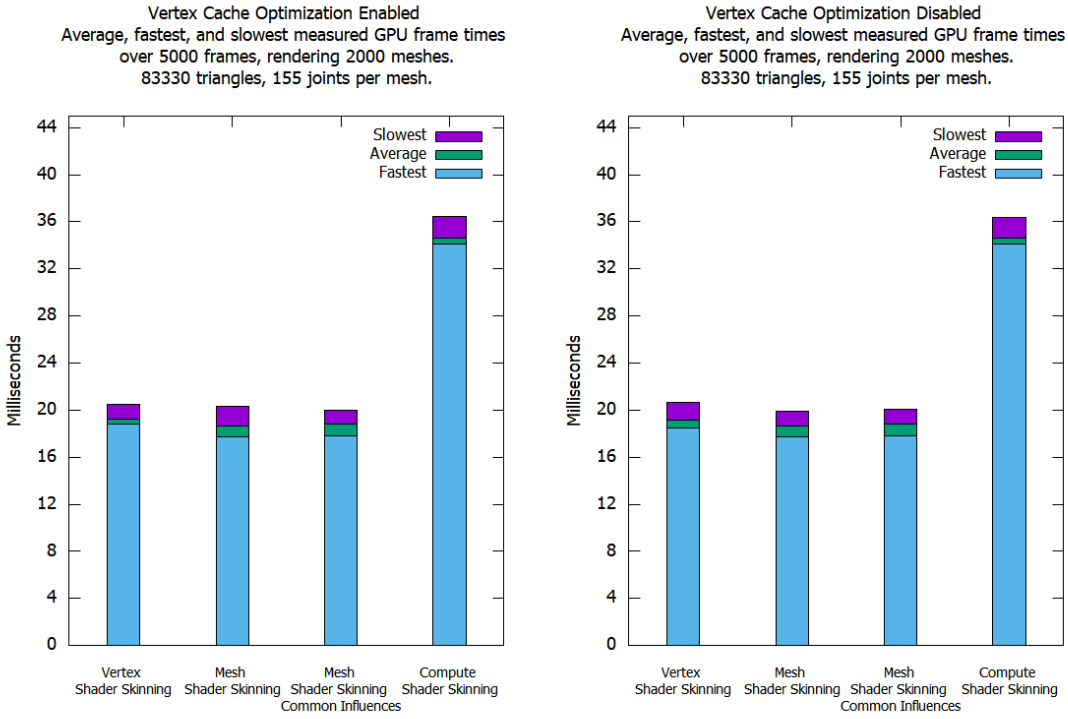

Figure 5.2: A comparison of all implementations rendering the complex mesh, with and without vertex cache optimization.

be fetched is reduced significantly. The total time spent per meshlet is less in the case of the common influences algorithm. A comparison of the time spent processing each meshlet can be viewed in table 5.4. While the speed up is not as significant when rendering the complex mesh, there is still a significant increase in performance with the simple human mesh. Which has its meshlets processed in 22% less time compared to the naive implementation. This establishes the fact that while the novel solution is faster on a per meshlet basis this gain is lost due to the increased number of meshlets. However this gives the common influences variant a chance to catch up if the difference in meshlets is sufficiently small. This was observed in figure 5.1 where the novel implementation outperformed the naive meshlet solution by a significant margin. When the optimization is turned off the difference in meshlets is reduced from 86 to a mere 9 meshlets (as seen in table 5.2), this allows the novel method to outperform the naive implementation.

While this may seem promising it is ultimately only proof of the fact that more work needs to be done in order to reduce the number of meshlets produced by the common influences method. It should be assumed that any and all modern real-time rendering engines will attempt to optimize the vertex cache as much as possible and thus it is unrealistic to expect any real world gain from the novel method in its current iteration. The best choice at the moment would be to simply utilize the naive solution until more work can be done on the novel meshlet generation algorithm, or any other possible alternative that would provide fewer meshlets since that is the main bottleneck. Ultimately utilizing mesh shaders for skinning is in fact faster than

	Simple Mesh	Complex Mesh
MS	∼0.023 ms per meshlet	∼0.02 ms per meshlet
MS CI	∼0.018 ms per meshlet (∼22% less time)	∼0.017 ms per meshlet (∼15% less time)

Table 5.4: The time spent processing each meshlet.

simply using a vertex shader and with this knowledge more work may be conducted in the future.

5.1.3 Optimize Mesh Shaders

Mesh Shaders allow for optimization techniques which aren't possible with the traditional vertex pipeline. For instance the ability to cull triangles on a per meshlet basis inside the mesh shader, reducing the amount of geometry that is sent down the rendering pipeline. Instead of relying on the slower built in back-face culling provided by the API which occurs later in the pipeline, one can improve performance quite noticeably by doing the culling earlier. A comparison can be seen in figure 5.3 where each mesh is rendered with and without the optimization. The method for the culling utilizes the determinant of a 3x3 matrix and is very cheap to do, allowing for fast culling in the mesh shader.[14, p. 46] As made evident by table 4.2, the difference in time taken between no skinning and skinning a Vertex Shader is negligible to the point where skinning itself could be considered free. The improvement due to the per meshlet triangle culling indicates that the skinning process itself is in fact no the main bottleneck of the shader but in fact the output to the rasterizer. By culling triangles early, fewer of triangles are sent to the rasterizer resulting in less work, thus the shorter frame time. It was observed that skinning the simple mesh took 12% less time with naive Mesh Shading compared to Vertex Shader skinning and around 10% less time with the Common Influences method. The complex mesh saw greater improvement with the naive implementation taking 21% less time and lastly the novel variant taking approximately 17% less time in comparison to Vertex Shader skinning. However it is worth noting that the back face culling optimization is not limited to skinning and is applicable to all geometry.

Initially the novel common influences mesh shading implementation relied on two separate passes. One pass for rendering the optimal meshlets and a second one for rendering the sub-optimal meshlets. This proved to be cumbersome to implement and maintain, and was later replaced by a single pass which could process both kinds of meshlets. Starting a second render pass does not incur a major penalty but it is still desirable to keep draw calls to a minimum (Fewer draw calls means fewer pipelines being created on the GPU) and by merging the two passes the number of draw calls was halved thanks to a very convenient realization. An unsigned 8 bit integer is used by the meshlet to count how many triangles it holds. The maximum value that 8 bits can hold is 255, and the maximum number of triangles a meshlet can have as per Nvidia's specification is 126. Allowing for the most significant to be set in order to denote that a meshlet is in fact an optimal meshlet as opposed to a sub-optimal one. The most significant bit can never be set by the meshlet itself while counting the number of triangles since as stated previously, the max is 126 and thus only 7 out of 8 bits are at most used at any given moment. This conveniently grants

the ability of checking the most significant bit in the shader in order to determine which path to take in the shader, thus allowing for easy branching in the shader and the ability to merge the two passes into one. Prior to this realization there was no way of discerning if a meshlet was of the good or bad kind without adding additional information to the meshlet itself and thus increasing the memory use. This way the additional information is stored effectively for free inside the triangle count variable of the meshlet.

Due to the compute-like nature of Mesh Shaders, there are many other possibilities to further optimize the shaders. Such as implementing per meshlet frustum culling. This would allow for the ability to effectively cull only parts of a skeletal mesh that are outside of the view frustum, something that is not normally possible when dealing with skeletal meshes which are traditionally difficult to cull. This due to the fact that a skeletal mesh can stretch far beyond its bounding box depending on the skeletal mesh in question so having only a single bounding box means that the mesh can either be rendered or culled fully. By culling on a per meshlet basis one could for example cull only the lower half of a mesh which is out of view while still successfully skinning the rest of it. As established earlier the number of meshlets plays a significant part in how efficient a Mesh Shading pipeline is so in instances where only part of a skeletal mesh is visible the Mesh Shading pipeline will only skin the visible part while a non-Mesh Shader based implementation would skin the entire mesh regardless of partial visibility.

5.1.4 Verify Common Influences Implementation Optimization

In order to verify the effectiveness of the proposed optimization with the common influences method. A final comparison must be made where the advantage of the per meshlet joint influences listing is disabled. When the optimization is disabled the shader uses a *fat* vertex as opposed to the slimmed down variant which doesn't contain the joint id information. A comparison can be seen in figure 5.4 where it becomes evident that the proposed optimization does indeed have a noticeable effect on performance.

5.2 Studying Memory Usage

The memory usage of each implementation presented in figure 4.4 is very straightforward and shows no unexpected behavior. Unsurprisingly the size of the buffers used play a part in the overall amount of memory used. The Vertex Shader implementation requires the fewest number of buffers, consisting only of a vertex and index buffer for the mesh itself, along with a buffer for the animation data. This straight forward approach yields a low amount of memory required as evident by the graph. In the case of the compute shader implementation it is once again the worst of the implementations. Due to the nature of the compute shader implementation, it must write to an out buffer so that the secondary graphics pass has access to the data and thus it consumes the most memory as a whole. Lastly the Mesh Shader implementations which are similar in design to a compute shader, also require a greater

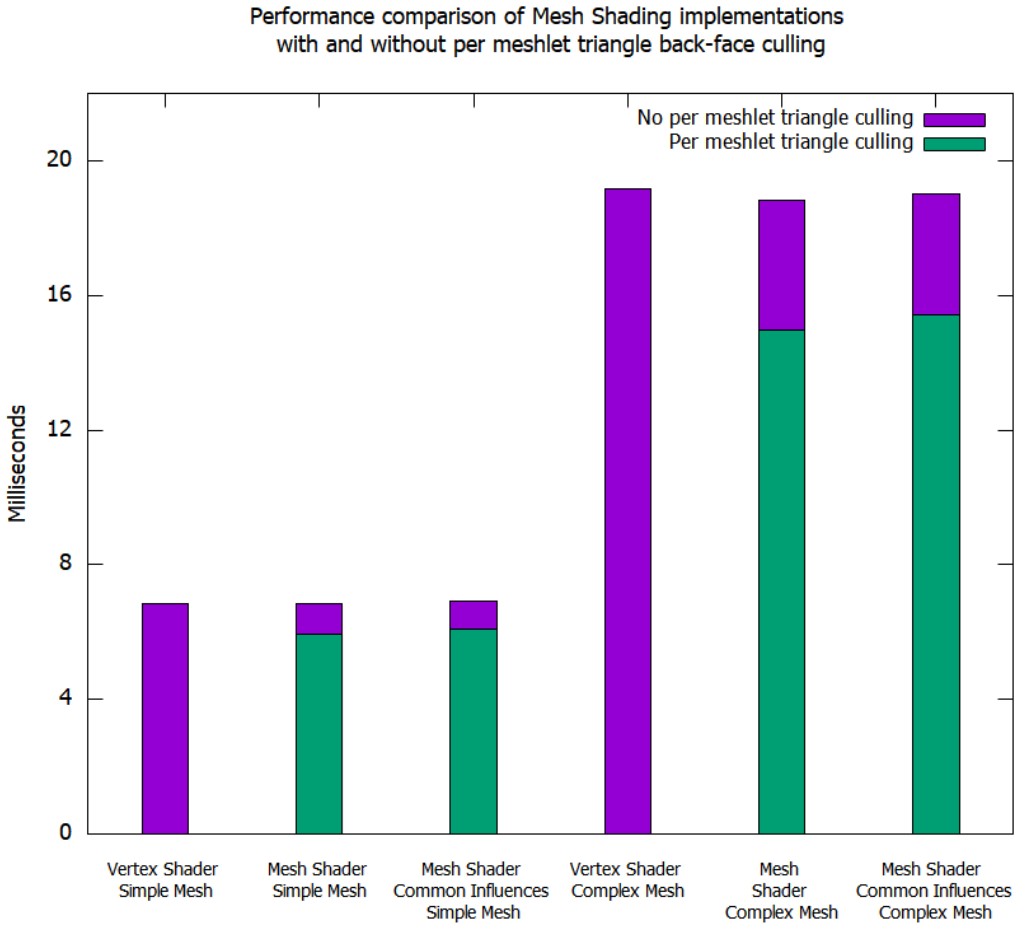

Figure 5.3: A comparison of the performance gain when using per meshlet triangle back-face culling.

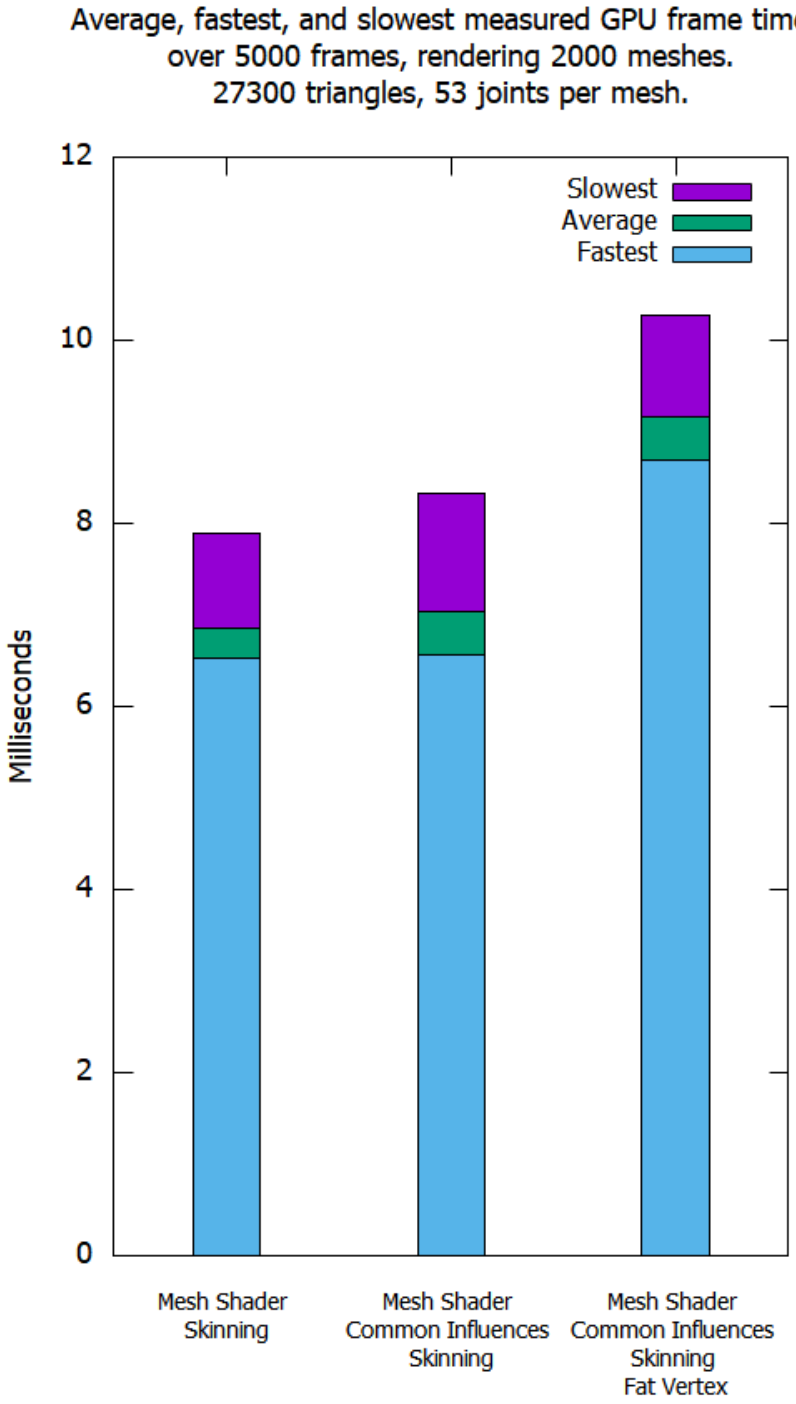

Figure 5.4: A comparison of the performance when using the optimized common influences method as opposed to using a fat vertex containing more data.

number of buffers. However instead of requiring an out buffer they can immediately output to a fragment shader as expected, with the key distinction being that they require meshlet buffers in order to function. A meshlet buffer contains information regarding all meshlets that are going to be rendered. A meshlet requires a fair bit of data since each meshlet has a list of vertices and a local index buffer defining how it should form triangles, along with the total triangle and vertex count of the meshlet. This will incur greater memory usage than a simple traditional graphics pipeline, but it still utilizes less memory compared to a compute shader due to it not requiring an out buffer for a secondary pass.

Lastly the Mesh Shader implementation utilizing common influences during the meshlet generation process utilizes less memory compared to the naive meshlet solution. This was expected due to the fact that less data is required on a per vertex basis. The list defining which joints affect a vertex is no longer stored in the vertex itself and is instead stored inside the meshlet, resulting in a small improvement over the naive solution.

5.3 Evaluating Accuracy

Despite the differences between each implementation, the underlying skinning procedure is still identical. This is why the raw data of each implementation is identical, as made evident by table 4.1 where the skinned position data is the same across all variants. Unless the data is manipulated elsewhere prior to being sent to the GPU the output will always remain identical, assuming the skinning calculations also remain identical in the shader.

Chapter 6

Conclusions and Future Work

In this thesis a novel method of skinning has been proposed in which Mesh Shaders are a key component. Two Mesh Shader based implementations were created, one utilizing said novel method and another naive implementation which divides the mesh in a naive manner instead of taking skinning information into account. Along these implementations several others were created for comparison including a traditional graphics pipeline and a compute shader pipeline. Lastly all implementations were given a variant where a depth and skinning pre-pass was utilized in order to determine if any implementation performs better when data must be reused.

While the results are not a resounding success regarding the novel common influences meshlet generation, Mesh Shading is in fact noticeably faster at skinning a skeletal mesh when optimization techniques unique to Mesh Shaders are employed. However without these optimizations the Mesh Shader based implementations are generally on par with the fastest of the previous implementations at most. The proposed novel solution is slightly slower due to producing more meshlets but not linearly slower which proves that the suggested optimization is valid, but more work must be conducted in order to reduce the number of meshlets generated. As made evident during testing, the novel solution may outperform the naive implementation if the difference in meshlets is sufficiently small due to it being faster on a per meshlet basis. It is worth keeping in mind that the Mesh Shader implementations are highly dependent on the mesh being used. This is especially true for the novel method which is directly impacted by how a mesh's weights are painted by an artist in an external program. If the skinning is done poorly it may produce a much worse result at run-time. For instance if the skinned mesh lacks any nicely defined regions such as a chest area where most vertices are affected by the same joints, then optimal meshlets will be fewer than those that are sub-optimal. An ideal scenario for the future would be to perform these same tests with an improved novel implementation and do so on a much larger set of meshes. Having a larger sampling of meshes would allow one to potentially determine the optimal kind of mesh that could be used in conjunction with the novel solution in order to produce the best results. Another shortcoming of the tests was the fact that only four weights were used per vertex. The justification for this was that due to four weights per vertex being the most common for video games it should also be the amount of weights used for these tests. While the reasoning is still sound, it may have been interesting to run these tests with five or more weights as well. For instance, Unreal Engine 4 supports up to eight

weights per vertex.[1] Developing a more extreme case with eight weights per vertex would mean that up to eight matrices would have to be fetched per vertex and since the novel common influences method requires only one fetch for an entire meshlet, the performance gains could be potentially much greater than they have been than now. It would also be interesting to compare the new pipeline to the mesh clustering method as described in the related work section. Mesh clustering is effectively the same concept as meshlets minus the hardware acceleration and comparing the two may be an interesting way of determining the usefulness of the special hardware required for Mesh Shaders to function.

Despite all of this, Mesh Shading does perform significantly better than Compute Shader skinning which is often employed in contemporary game engines. Since Mesh Shaders may be used for generic Compute work, it may be viable to implement Mesh Shader skinning as an alternative if the required hardware is available.

Memory usage as a whole remains very small due to modern GPUs generally having very large amounts of memory. However it is still worth taking into consideration and the novel solution does lower the memory footprint by a small amount but the reduction but the algorithm is not worth pursuing just for reduced memory usage.

As for how the naive and novel implementations fare when a pre-pass is utilized, the performance gains are mostly lost but they do still perform marginally better than the Vertex Shader implementation.

The Mesh Shader pipeline is still a very new concept and has yet to be used in a real product. The whole pipeline is still very much uncharted territory and thus quite difficult to work with due to the limited information available. The new pipeline is still only an extension to existing APIs and not actually a part of any core and thus not all vendors support the technology. This has only been a first step into evaluating the potential of the new pipeline. For this reason, much more work must be conducted in order to determine the ultimate usefulness of the Mesh Shader pipeline in regards to skinning meshes.

[1] Confirmed by an Epic Games employee `https://answers.unrealengine.com/questions/8117/bone-limitations.html`, Accessed 14th of June 2019

References

[1] Duane C. Abbey, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition, 4th Edition*. A K Peters/CRC Press, 4 edition, 2018.

[2] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional, first edition, 2004.

[3] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, 2003.

[4] Tom Forsyth. Linear-speed vertex cache optimisation, 2016. `https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html`, Accessed: April 26th, 2019.

[5] Alec Jacobson, Ilya Baran, Ladislav Kavan, Jovan Popović, and Olga Sorkine. Fast automatic skinning transformations. *ACM Trans. Graph.*, 31(4):77:1–77:10, July 2012.

[6] Sagi Katz and Ayellet Tal. Hierarchical mesh decomposition using fuzzy clustering and cuts. *ACM Trans. Graph.*, 22(3):954–961, July 2003.

[7] YoungBeom Kim and JungHyun Han. Bulging-free dual quaternion skinning. *Computer Animation and Virtual Worlds*, 25(3-4):323–331, 2014.

[8] Chrristoph Kubisch. Introduction to turing mesh shaders, 2018. `https://devblogs.nvidia.com/introduction-turing-mesh-shaders/` Accessed: March 11th, 2019.

[9] Binh Le and Jessica Hodgins. Real-time skeletal skinning with optimized centers of rotation. *ACM Transactions on Graphics (TOG)*, 35(4):1–10, 2016.

[10] N. Magnenat-Thalmann, R. Laperrière, and D. Thalmann. Joint-dependent local deformations for hand animation and object grasping. In *Proceedings on Graphics Interface '88*, pages 26–33, Toronto, Ont., Canada, Canada, 1988. Canadian Information Processing Society.

[11] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 1 edition, 2007.

[12] Graham Sellers. *Vulkan™ Programming Guide*. Addison-Wesley Professional, 1 edition, 2016.

[13] János Turanszkij. Skinning in a compute shader, Sep 2017. `https://wickedengine.net/2017/09/09/skinning-in-compute-shader/`, Accessed: March 21st, 2019.

[14] Graham Wihlidal. Optimizing the graphics pipeline with compute, 2018. https://www.ea.com/frostbite/news/optimizing-the-graphics-pipeline-with-compute, Accessed: March 21st, 2019. Alternative version with comments: https://frostbite-wp-prd.s3.amazonaws.com/wp-content/uploads/2016/03/29204330/GDC_2016_Compute.pdf.

Appendix A

Supplemental Information

Listing A.1: Source code for meshlet generation.

```
//This function is called directly with the full indices list when generating meshlets naively.
void Mesh::GenerateMeshlets(const std::vector<uint32_t> &MeshletIndices, bool Optimal)
{
    if (MeshletIndices.empty())
    {
        return;
    }

    Meshlet NewMeshlet = {};
    std::vector<uint32_t> MeshletVertices(_Vertices.size(), 0xff);

    for (size_t i = 0; i < MeshletIndices.size(); i += 3)
    {
        uint32_t IndexA = MeshletIndices[i + 0];
        uint32_t IndexB = MeshletIndices[i + 1];
        uint32_t IndexC = MeshletIndices[i + 2];

        uint32_t& AVertex = MeshletVertices[IndexA];
        uint32_t& BVertex = MeshletVertices[IndexB];
        uint32_t& CVertex = MeshletVertices[IndexC];

        if (NewMeshlet.VertexCount + (AVertex == 0xff) + (BVertex == 0xff) +
        (CVertex == 0xff) > MeshletMaxVertices || NewMeshlet.TriangleCount >= 126)
        {
            if (Optimal)
            {
                NewMeshlet.TriangleCount |= 0x80; //Set most significant bit to 1, indicating that this is an optimal meshlet. This bit is later checked in the shader in order to determine branching.
            }
            _Meshlets.push_back(NewMeshlet);
            for (size_t j = 0; j < NewMeshlet.VertexCount; ++j)
            {
                MeshletVertices[NewMeshlet.Vertices[j]] = 0xff;
            }

            NewMeshlet = {};
        }

        if (AVertex == 0xff)
        {
            AVertex = NewMeshlet.VertexCount;
            NewMeshlet.Vertices[NewMeshlet.VertexCount++] = IndexA;
        }

        if (BVertex == 0xff)
        {
            BVertex = NewMeshlet.VertexCount;
            NewMeshlet.Vertices[NewMeshlet.VertexCount++] = IndexB;
        }

        if (CVertex == 0xff)
        {
            CVertex = NewMeshlet.VertexCount;
            NewMeshlet.Vertices[NewMeshlet.VertexCount++] = IndexC;
        }

        NewMeshlet.Indices[NewMeshlet.TriangleCount * 3 + 0] = AVertex;
        NewMeshlet.Indices[NewMeshlet.TriangleCount * 3 + 1] = BVertex;
        NewMeshlet.Indices[NewMeshlet.TriangleCount * 3 + 2] = CVertex;
        NewMeshlet.TriangleCount++;
    }

    if (NewMeshlet.TriangleCount)
    {
        if (Optimal)
        {
            NewMeshlet.TriangleCount |= 0x80;
        }
        _Meshlets.push_back(NewMeshlet);
    }
}
```

Listing A.2: Source code for the novel common influences meshlet generation.

```
1
2  void Mesh::BuildMeshletsCommonInfluences(DeviceHandler* Device)
3  {
4      std::map<uint64_t, std::vector<uint32_t>> MeshletIndicesMap;
5      uint64_t RemainderKey = UINT64_MAX_SIZE;
6
7      for (size_t i = 0; i < _Indices.size(); i += 3)
8      {
9          uint32_t IndexA = _Indices[i + 0];
10         uint32_t IndexB = _Indices[i + 1];
11         uint32_t IndexC = _Indices[i + 2];
12
13
14         if (_Vertices[IndexA] == _Vertices[IndexB] && _Vertices[IndexA] == _Vertices[IndexC])
15         {
16             //Triangle has identical influences
17             uint64_t Key = (uint64_t)(
18                 ((uint64_t)_Vertices[IndexA]._JointIDs[0] << 48) |
19                 ((uint64_t)_Vertices[IndexA]._JointIDs[1] << 32) |
20                 ((uint64_t)_Vertices[IndexA]._JointIDs[2] << 16) |
21                 ((uint64_t)_Vertices[IndexA]._JointIDs[3] << 0));
22
23             MeshletIndicesMap[Key].push_back(IndexA);
24             MeshletIndicesMap[Key].push_back(IndexB);
25             MeshletIndicesMap[Key].push_back(IndexC);
26         }
27         else
28         {
29             //Triangle does not have identical influences across vertices, put triangle in a
       separate meshlet
30             MeshletIndicesMap[RemainderKey].push_back(IndexA);
31             MeshletIndicesMap[RemainderKey].push_back(IndexB);
32             MeshletIndicesMap[RemainderKey].push_back(IndexC);
33         }
34     }
35
36     //Put triangles with identical influences into the Remainder group if they are very few,
37     //in order to prevent meshlets with low amount of triangles from being created
38     for (auto& [Key, Indices] : MeshletIndicesMap)
39     {
40         if (Indices.size() < MeshletMinimumVertexCount)
41         {
42             MeshletIndicesMap[RemainderKey].insert(MeshletIndicesMap[RemainderKey].end(),
43             Indices.begin(), Indices.end());
44             Indices.clear();
45         }
46     }
47
48     for (const auto& [Key, MeshletIndices] : MeshletIndicesMap)
49     {
50         if (Key == RemainderKey)
51         {
52             continue;
53         }
54
55         GenerateMeshlets(MeshletIndices, true); //Generate optimal meshlets
56     }
57
58     GenerateMeshlets(MeshletIndicesMap.at(RemainderKey), false); //Generate sub optimal meshlets
```

Listing A.3: Shader source code for naive meshlet rendering

```
1
2  #version 450
3  const int MAX_JOINTS = 255;
4
5  #extension GL_NV_mesh_shader: require
6  #extension GL_EXT_shader_16bit_storage: require
7  #extension GL_EXT_shader_8bit_storage: require
8
9  #extension GL_KHR_shader_subgroup_basic: enable
10 #extension GL_KHR_shader_subgroup_ballot : enable
11 #extension GL_KHR_shader_subgroup_arithmetic : enable
12
13 layout(local_size_x = 32) in;
14 layout(triangles, max_vertices = 64, max_primitives = 126) out;
15
16 //Custom vector 3 due to memory alignment issues
17 struct vector3
18 {
19    float x;
20    float y;
21    float z;
22 };
23
24 struct Vertex
25 {
26    vector3 Pos;
27    vector3 Nor;
28    float Weights[4];
29    int JointIDs[4];
30 };
31
32 struct Meshlet
33 {
34    uint Vertices[64];
35    uint8_t Indices[126*3]; // Up to 126 triangles
36    uint8_t TriangleCount;
37    uint8_t VertexCount;
38 };
39
40 layout(set = 0, binding = 0) buffer MeshletsIn
41 {
42    Meshlet MeshletIn[];
43 };
44
45 layout(set = 0, binding = 1) buffer VerticesIn
46 {
47    Vertex VertexIn[];
48 };
49
50 layout(set = 1, binding = 0) uniform JointBuffer
51 {
52    mat4 MVP;
53    mat4 JointList[MAX_JOINTS];
54 };
55
56 layout(location = 0) out vec3 Normal[];
57 layout(location = 1) out vec3 Color[];
58
59 void main()
60 {
61    uint Id = gl_WorkGroupID.x;
62    uint TId = gl_LocalInvocationID.x;
63
64    uint VertexCount = uint(MeshletIn[Id].VertexCount);
65    for(uint i = TId; i < VertexCount; i += 32)
66    {
67       uint Vi = MeshletIn[Id].Vertices[i];
68
69       mat4 JointTransform = mat4(1);
70       JointTransform = JointList[VertexIn[Vi].JointIDs[0]] * VertexIn[Vi].Weights[0];
71       JointTransform += JointList[VertexIn[Vi].JointIDs[1]] * VertexIn[Vi].Weights[1];
72       JointTransform += JointList[VertexIn[Vi].JointIDs[2]] * VertexIn[Vi].Weights[2];
73       JointTransform += JointList[VertexIn[Vi].JointIDs[3]] * VertexIn[Vi].Weights[3];
74
75       vec3 Position = vec3(VertexIn[Vi].Pos.x, VertexIn[Vi].Pos.y, VertexIn[Vi].Pos.z);
76
77       vec4 ProjectedPosition = MVP * JointTransform * vec4(Position, 1.0);
78
79       gl_MeshVerticesNV[i].gl_Position = ProjectedPosition;
80
81       Color[i] = vec3(0.5, 0.5, 0.5);
82
83       Normal[i] = normalize(vec3(JointTransform * vec4(VertexIn[Vi].Nor.x, VertexIn[Vi].Nor.y,
     VertexIn[Vi].Nor.z, 1.0)));
84    }
85
86 #if 0
87    //Old path without per meshlet triangle culling
88    uint IndexCount = uint(MeshletIn[Id].TriangleCount) * 3;
89    for(uint i = TId; i < IndexCount; i += 32)
90    {
91       gl_PrimitiveIndicesNV[i] = uint(MeshletIn[Id].Indices[i]);
92    }
93    if (TId == 0)
```

```
94      {
95          gl_PrimitiveCountNV = uint(MeshletIn[Id].TriangleCount);
96      }
97  #else
98      const uint TriangleCount = uint(MeshletIn[Id].TriangleCount);
99      uint LocalPrimitiveCount = 0;
100     for (uint i = TId; i < TriangleCount; i += 32)
101     {
102         uint I0 = uint(MeshletIn[Id].Indices[i * 3 + 0]);
103         uint I1 = uint(MeshletIn[Id].Indices[i * 3 + 1]);
104         uint I2 = uint(MeshletIn[Id].Indices[i * 3 + 2]);
105
106         vec3 V0 = gl_MeshVerticesNV[I0].gl_Position.xyw;
107         vec3 V1 = gl_MeshVerticesNV[I1].gl_Position.xyw;
108         vec3 V2 = gl_MeshVerticesNV[I2].gl_Position.xyw;
109         float Determinant = determinant(mat3(V0, V1, V2));
110         bool IsVisible = Determinant < 0.0f; // varying per lane
111
112         uvec4 IsVisibleMask = subgroupBallot(IsVisible); // result is uniform per warp
113         uint LocalIndex = subgroupBallotExclusiveBitCount(IsVisibleMask); // varying per lane
114
115         if (IsVisible)
116         {
117             uint WritePos = LocalPrimitiveCount + LocalIndex; // compacted write index
118
119             gl_PrimitiveIndicesNV[WritePos * 3 + 0] = I0;
120             gl_PrimitiveIndicesNV[WritePos * 3 + 1] = I1;
121             gl_PrimitiveIndicesNV[WritePos * 3 + 2] = I2;
122         }
123
124         // count all lanes that considered primitive to be visible
125         LocalPrimitiveCount += subgroupBallotBitCount(IsVisibleMask); // LocalPrimitiveCount
      contains same value in each lane after this
126     }
127
128     if(TId == 0)
129     {
130         gl_PrimitiveCountNV = LocalPrimitiveCount;
131     }
132
133 #endif
134 }
```

Listing A.4: Shader source code for novel common influences meshlet rendering

```
1
2  #version 450
3  const int MAX_JOINTS = 255;
4
5  #extension GL_NV_mesh_shader : require
6  #extension GL_EXT_shader_16bit_storage : require
7  #extension GL_EXT_shader_8bit_storage : require
8
9  #extension GL_KHR_shader_subgroup_basic : enable
10 #extension GL_KHR_shader_subgroup_ballot : enable
11 #extension GL_KHR_shader_subgroup_arithmetic : enable
12
13 layout(local_size_x = 32) in;
14 layout(triangles, max_vertices = 64, max_primitives = 126) out;
15
16 #define USE_FAT_VERTEX 0
17
18 //Custom vector 3 due to memory alignment issues
19 struct vector3
20 {
21    float x;
22    float y;
23    float z;
24 };
25
26 struct Vertex
27 {
28    vector3 Pos;
29    vector3 Nor;
30    float Weights[4];
31 #if USE_FAT_VERTEX
32    int JointIDs[4];
33 #endif
34 };
35
36 struct Meshlet
37 {
38    uint Vertices[64];
39    uint8_t Indices[126 * 3];
40    uint8_t TriangleCount;
41    uint8_t VertexCount;
42 };
43
44 struct IDs
45 {
46    uint8_t x;
47    uint8_t y;
48    uint8_t z;
49    uint8_t w;
50 };
51
52 layout(set = 0, binding = 0) buffer MeshletsIn
53 {
54    Meshlet MeshletIn[];
55 };
56
57 layout(set = 0, binding = 1) buffer VerticesIn
58 {
59    Vertex VertexIn[];
60 };
61
62 layout(set = 0, binding = 2) buffer JointIDsIn
63 {
64    IDs JointIDs[];
65 };
66
67 layout(set = 1, binding = 0) uniform JointBuffer
68 {
69    mat4 MVP;
70    mat4 JointList[MAX_JOINTS];
71 };
72
73 layout(location = 0) out vec3 Normal[];
74 layout(location = 1) out vec3 Color[];
75
76 void main()
77 {
78    uint Id = gl_WorkGroupID.x;
79    uint TId = gl_LocalInvocationID.x;
80
81    uint VertexCount = uint(MeshletIn[Id].VertexCount);
82
83    const bool CommonJointInfluences = (uint(MeshletIn[Id].TriangleCount) & 0x80) == 0x80;
84
85    if (CommonJointInfluences)
86    {
87 #if !USE_FAT_VERTEX
88
89       uint VertexIndex = MeshletIn[Id].Vertices[0];
90       uint JointID0 = int(JointIDs[VertexIndex].x);
91       uint JointID1 = int(JointIDs[VertexIndex].y);
92       uint JointID2 = int(JointIDs[VertexIndex].z);
93       uint JointID3 = int(JointIDs[VertexIndex].w);
94
```

```
95        //All vertices affected by the same four joints, fetch their matrices once
96        mat4 Joint0 = JointList[JointID0];
97        mat4 Joint1 = JointList[JointID1];
98        mat4 Joint2 = JointList[JointID2];
99        mat4 Joint3 = JointList[JointID3];
100 #endif
101
102       for (uint i = TId; i < VertexCount; i += 32)
103       {
104           uint Vi = MeshletIn[Id].Vertices[i];
105
106           mat4 JointTransform = mat4(1);
107
108 #if USE_FAT_VERTEX
109           JointTransform =  JointList[VertexIn[Vi].JointIDs[0]] * VertexIn[Vi].Weights[0];
110           JointTransform += JointList[VertexIn[Vi].JointIDs[1]] * VertexIn[Vi].Weights[1];
111           JointTransform += JointList[VertexIn[Vi].JointIDs[2]] * VertexIn[Vi].Weights[2];
112           JointTransform += JointList[VertexIn[Vi].JointIDs[3]] * VertexIn[Vi].Weights[3];
113 #else
114           JointTransform =  Joint0 * VertexIn[Vi].Weights[0];
115           JointTransform += Joint1 * VertexIn[Vi].Weights[1];
116           JointTransform += Joint2 * VertexIn[Vi].Weights[2];
117           JointTransform += Joint3 * VertexIn[Vi].Weights[3];
118 #endif
119           vec3 Position = vec3(VertexIn[Vi].Pos.x, VertexIn[Vi].Pos.y, VertexIn[Vi].Pos.z);
120
121           gl_MeshVerticesNV[i].gl_Position = MVP * JointTransform * vec4(Position, 1.0);
122
123           Color[i] = vec3(0.5, 0.5, 0.5);
124
125           Normal[i] = normalize(vec3(JointTransform * vec4(VertexIn[Vi].Nor.x,
126           VertexIn[Vi].Nor.y, VertexIn[Vi].Nor.z, 1.0)));
127       }
128   }
129   else
130   {
131       for (uint i = TId; i < VertexCount; i += 32)
132       {
133           uint Vi = MeshletIn[Id].Vertices[i];
134
135           mat4 JointTransform = mat4(1);
136
137 #if USE_FAT_VERTEX
138           JointTransform = JointList[VertexIn[Vi].JointIDs[0]] * VertexIn[Vi].Weights[0];
139           JointTransform += JointList[VertexIn[Vi].JointIDs[1]] * VertexIn[Vi].Weights[1];
140           JointTransform += JointList[VertexIn[Vi].JointIDs[2]] * VertexIn[Vi].Weights[2];
141           JointTransform += JointList[VertexIn[Vi].JointIDs[3]] * VertexIn[Vi].Weights[3];
142 #else
143           uint Joint0 = int(JointIDs[Vi].x);
144           uint Joint1 = int(JointIDs[Vi].y);
145           uint Joint2 = int(JointIDs[Vi].z);
146           uint Joint3 = int(JointIDs[Vi].w);
147
148           JointTransform = JointList[Joint0] * VertexIn[Vi].Weights[0];
149           JointTransform += JointList[Joint1] * VertexIn[Vi].Weights[1];
150           JointTransform += JointList[Joint2] * VertexIn[Vi].Weights[2];
151           JointTransform += JointList[Joint3] * VertexIn[Vi].Weights[3];
152 #endif
153           vec3 Position = vec3(VertexIn[Vi].Pos.x, VertexIn[Vi].Pos.y, VertexIn[Vi].Pos.z);
154
155           gl_MeshVerticesNV[i].gl_Position = MVP * JointTransform * vec4(Position, 1.0);
156
157           Color[i] = vec3(0.5, 0.5, 0.5);
158
159           Normal[i] = normalize(vec3(JointTransform * vec4(VertexIn[Vi].Nor.x,
160           VertexIn[Vi].Nor.y, VertexIn[Vi].Nor.z, 1.0)));
161       }
162   }
163
164 #if 0
165   uint TriCount = 0x7F & uint(MeshletIn[Id].TriangleCount);
166   uint IndexCount = TriCount * 3;
167   for (uint i = TId; i < IndexCount; i += 32)
168   {
169       gl_PrimitiveIndicesNV[i] = uint(MeshletIn[Id].Indices[i]);
170   }
171
172   if (TId == 0)
173   {
174       gl_PrimitiveCountNV = TriCount;
175   }
176 #else
177
178   uint TriangleCount = 0x7F & uint(MeshletIn[Id].TriangleCount);
179   uint LocalPrimitiveCount = 0;
180   for (uint i = TId; i < TriangleCount; i += 32)
181   {
182       uint I0 = uint(MeshletIn[Id].Indices[i * 3 + 0]);
183       uint I1 = uint(MeshletIn[Id].Indices[i * 3 + 1]);
184       uint I2 = uint(MeshletIn[Id].Indices[i * 3 + 2]);
185
186 #if 1
187       vec3 V0 = gl_MeshVerticesNV[I0].gl_Position.xyw;
188       vec3 V1 = gl_MeshVerticesNV[I1].gl_Position.xyw;
189       vec3 V2 = gl_MeshVerticesNV[I2].gl_Position.xyw;
190       float Determinant = determinant(mat3(V0, V1, V2));
```

```
191         bool IsVisible = Determinant < 0.0f; // varying per lane
192 #else
193         vec4 v0 = gl_MeshVerticesNV[I0].gl_Position;
194         vec4 v1 = gl_MeshVerticesNV[I1].gl_Position;
195         vec4 v2 = gl_MeshVerticesNV[I2].gl_Position;
196         v0.xyz /= v0.w;
197         v1.xyz /= v1.w;
198         v2.xyz /= v2.w;
199         bool IsVisible = cross(v1.xyz   v0.xyz, v2.xyz   v0.xyz).z < 0;
200 #endif
201 
202         uvec4 IsVisibleMask = subgroupBallot(IsVisible); // result is uniform per warp
203         uint LocalIndex = subgroupBallotExclusiveBitCount(IsVisibleMask); // varying per lane
204 
205         if (IsVisible)
206         {
207             uint WritePos = LocalPrimitiveCount + LocalIndex; // compacted write index
208 
209             gl_PrimitiveIndicesNV[WritePos * 3 + 0] = I0;
210             gl_PrimitiveIndicesNV[WritePos * 3 + 1] = I1;
211             gl_PrimitiveIndicesNV[WritePos * 3 + 2] = I2;
212         }
213 
214         // count all lanes that considered primitive to be visible
215         LocalPrimitiveCount += subgroupBallotBitCount(IsVisibleMask); // LocalPrimitiveCount
      contains same value in each lane after this
216     }
217 
218     if (TId == 0)
219     {
220         gl_PrimitiveCountNV = LocalPrimitiveCount;
221     }
222 
223 
224 #endif
225 }
```

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden