TUM SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik: Games Engineering

Hardware-specific Rendering of Hierarchical Meshlets

Jonas M. Lehmann

TUM SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik: Games Engineering

Hardware-specific Rendering of Hierarchical Meshlets

Hardware-spezifisches Rendering von hierachischen Meshlets

Bearbeiter:	Jonas M. Lehmann
Themensteller:	Prof. Dr. Rüdiger Westermann
Betreuer:	M.Sc. Christoph Neuhauser
Eingereicht am:	15.05.2024

Ich versichere hiermit, dass ich die von mir eingereichte Masterarbeit in Informatik: Games Engineering selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

München, 15.05.2024 Jonas M. Lehmann

Acknowledgments

I want to thank my advisor, Christoph Neuhauser as well as my supervisor Prof. Dr. Rüdiger Westermann to provide me with the opportunity to dive into such an interesting topic.

I would further like to thank all friends and family who gave encouragement and support during the creation period of this work

Abstract

In this thesis, a wide-ranging analysis of the components for building a hierarchical meshlet rendering system on modern graphics processing units is conducted.

Models for predicting the demands on computational as well as memory storage and bandwidth requirements are built. This entails a model for predicting processing efficiency for hardware accelerated triangle rasterization which is validated experimentally.

Furthermore, this thesis presents a novel algorithm – the "Mountain-Cleaver" algorithm – for meshlet generation. This is a specialized solver for the NP-complete graph partitioning of meshlet generation. The algorithm is demonstrated to construct meshlets more suitable for hierarchical rendering than many other common approaches.

The additional techniques and algorithms discussed and presented in this thesis, accumulate to give recipes for crafting efficient hierarchical meshlet rendering applications tuned to the needs of the application and the capabilities of the hardware.

Table of Contents

1. Introduction

The evolution of computer graphics brings along an ever increasing demand for the geometric complexity to be processed interactively. For a computer system to be able to provide the demanded level of interactivity, a Graphics Processing Unit (GPU) is employed. The hardware of GPUs has evolved from being simple accelerators for a fixed function pipeline to be powerful co-processors housing a massively parallel architecture. The evolution of GPUs also entailed the ability for general purpose compute, making GPUs capable of more than accelerating graphics related tasks. However, growth in computational demands tends to outpace the rate at which hardware becomes more capable. For an application to keep up with increasing demands, more efficient usage of the available resources is required.

Efficiency is measured by how much work is done to achieve a result. It follows logically that an increase in efficiency can only be achieved by performing less work for the same result. The result of an image generation / rendering operation is the picture that the user of the application will be able to see. Rendering 3D graphics efficiently therefore means to only work on geometry that will be visible.

The fundamental primitive for geometry Graphics Cards work with is the triangle. The complexity of a 3D model can be measured by how many triangles it is composed of. 3D models can have upwards of several hundreds of thousands to millions of triangles that require processing. When a virtual world is composed of multiple of those high complexity models, even modern graphics processing units will not be able to manage all while retaining the desired level of real-time interactivity.

There are already many established techniques to determine whether an instance of a 3D model could be potentially be seen and is therefore worth spending additional processing time on. For example, all triangles of an instance can skip processing if the instance is either "off screen", meaning not in the view frustum of the camera (frustum culling), or if the object is known to be fully occluded by another object (occlusion culling). Applying these techniques limits the set of potential work to the set of 3D models that are at least partially visible. While this already leads to a large reduction in the amount of triangles that need to be processed, in the traditional raster based rendering pipelines, partially visible models still require processing of all triangles even if only a fraction of the model is visible.

A finer granularity of visibility tests is needed for further removing invisible pieces of geometry. Finer granularity in this case involves breaking down the mesh of triangles that

represent the 3D model into smaller chunks called *meshlets* (also called triangle clusters). To every meshlets, culling techniques can then be applied to further filter out invisible or occluded pieces of geometry, further closing the gap between which geometry is visible versus which geometry processing time is spend on.

However, just because geometry is visible of the virtual camera does not mean it will also be visible to the user. The image the user will see is quantized into individual pixel each of which can only display one distinct color. Processing geometry that has no influence on any pixel of the final result could therefore also be skipped. This was traditionally achieved by switching the mesh of a 3D model to a representation with decreased geometric complexity / level of detail (LOD). The LOD of a model can be constructed and during application run time chosen such that the visual difference between the high complexity and the low complexity version is minimal up to being nonexistent.

As with culling techniques, applications of LOD where limited to the granularity of the model instance. The consequences of per instance granularity for level of detail are that some areas of image might be over-detailed, i.e. more geometry is present than necessary, while other areas could remain under-detailed. Solving the granularity issue by using meshlets is unfortunately not as straight forward as applying culling techniques. The reason for this is that when applied naively, meshlet of different levels of detail form visual "cracks" or "seams" at the boundary between. Those cracks introduce a perceptible error in the image presented to the user and should therefore be avoided.

Recent advances in meshlet rendering pipelines address the issue of cracks between levels of detail by introducing a specially crafted (LOD-)hierarchy of meshlets. Selecting a per meshlet level of detail that is seamless when rendered alongside other meshlets who select a different LOD-representation is made possible by utilizing the structure of this hierarchy. As demonstrated most prominently by Epic Game's Nanite system [1] in the Unreal Engine 5 game engine, hierarchical meshlet rendering can manage scenes with a geometric complexity in the high millions to billions of triangles on supported hardware.

Although the result achieved with Nanite are indeed impressive, the Nanite system as a whole targets primarily high-end hardware. However, the increase in work efficiency made possible by the techniques used for hierarchical meshlet rendering should in principle also apply lower end hardware such as GPUs found in mobile phones or laptop computers. In the following chapters of this thesis, an analysis of the required hardware capabilities for hierarchical meshlet rendering is conducted.

2. GPU Architecture

Core to finding an efficient utilization of graphics processing hardware is to develop an understanding of how this hardware operates on a fundamental level. While the exact details of a GPU's architecture differ from one hardware vendor to another as well as within different generations of the same GPU family [2] [3] [4] [5], there are some principles that have established themselves as a solid building block. These basic blocks are present in effectively all relevant graphics processors in some form or another.

2.1. Core components

Reducing the essence of a GPU down to the bare essentials, three core components with a specialized purpose present themselves:

- Computational Units

  These are the components of a GPU responsible for performing data transformations. Transforming data from one representation to another is in essence what all digital data processing does, so it is no surprise that GPUs contain components for this purpose.

  On any given GPU there are potentially many kinds of computational units. Some may be programmable and can be utilized for general purpose computations while others are fixed function and optimized to perform one specific transformation as efficiently as possible.

  Understanding the capabilities of the computational units is required to know how any operation can be executed in the most efficient way.

- Scheduler / Work Distributor

  The speciality of a GPU is its capacity for massively parallel processing. However, a collection of a large number of computational units is useless without some way to coordinate between the units. This is where the scheduler unit stands out as being responsible for distributing work items and generally managing when and how computations are executed.

Depending on how much work is expected to be executed, there may be a hierarchical arrangement of schedulers each responsible for a specific section of a GPU. For example, when a large quantity of work arrives, a coarse division among individual compute clusters may be done first with each packet in turn being managed by a cluster local scheduler.

Understanding how the scheduler might distributes the work given to him is important to know for when, how and in which form a package of work is best given to the GPU.

- Memory storage units & Memory Bus

  For a computation to be performed on a piece of data, this data first needs to be provided from somewhere. To ensure that the result of a data transformation can experience a further processing step, it needs to be stored in some place. These places are the memory storage units of the GPU and the transfer paths between storage and compute units is the memory bus system.

  Depending on which piece of data is needed where, different types of memory storage are used in different sections of the GPU. For example, integrated into a computational unit some small amount of data is stored in *registers* which are storage cells for directly providing the data and storing the (intermediate) results a compute instruction. Since register are limited in size, a memory unit with high capacity storage but comparatively high access latency – the GPU's "main memory" – is used to store larger pieces of data such as geometry and texture information as well as other data needed by the GPU.

  Understanding when and how data is located in which memory unit is key in structuring data such that computational units can be efficiently utilized and do not need to stall to wait for the completion of data transfer operations.

2.2. Specialized Components

Out of the three basic components, GPUs tend to have

- SIMD compute unit

  The core of the general purpose compute in modern GPUs consists of SIMD (Single Instruction, Multiple Data) processing units. These are computational units where the same sequence of instructions is applied to multiple work items in parallel. The instructions are in lock-step, meaning that at any given point in the instruction sequence, the exact same instruction is run on every work item.

A SIMD unit consists of a number of *Lanes*. The number of lanes per SIMD-unit – also referred to as its width - is the number of work items (or threads) that can be processed in parallel. During the execution of an instruction sequence, some lanes can be *masked out* as to only apply operations to specific elements and not to others.

The SIMD units are typically organized in clusters with a dispatcher unit handling work distribution within the cluster of SIMD units. SIMD units within such a cluster are often able to communicate efficiently with each other through the use of shared resources like local caches. Some architecture also support so called *horizontal* operations, that provide a way for lanes to exchange data within lanes of the same unit or to perform operations reduction operations on all data items within a SIMD unit.

- Memory Caches

  Accessing memory stored in main memory, which is often a form of SDRAM, is an operation with a very high latency compared to accessing data stored in registers. Ideally, all of the data for a computation would be kept in registers, but the limited number and size of registers does not permit this ideal. Since any piece of data has a high likelihood of being required for more than a single computation step, small amounts of high speed memory, called caches, are used to buffer results from reading or writing to main memory. While not as fast to access as registers, a cache has a significantly reduced latency compared to accesses to main memory. If a piece of data is requested and already resides in a cache, this is referred to as a *cache-hit*. If the data does not reside in the cache and must be pulled from main memory, it is is called a *cache-miss*.

  Caches operate on the granularity of a cache line instead of on individual bits and bytes. Because of this, when a certain byte of memory is requested, a whole cache line – typically 64 or 128 byte – is fetched from memory instead of only a single byte. This means an access to a different byte byte located close the first byte has a high likelihood of being a cache-hit and thus not requiring an additional memory transfer operation to occur. For optimal usage of resources, computations should aim to utilize as much of the memory present in a cache line as possible.

  In practice, caches have a limited size and thus often need to evict data / write data back to main memory, even if it is still needed for a later computation. However, for the purposes of the analysis conducted in this paper, it is assumed that any piece of data that could feasibly be cached will be. This means additional memory bandwidth costs due to cache eviction are not taken into consideration.

- Rasterizer

The rasterizer is a specialized component for testing which pixels are covered by triangles and generating work items accordingly. The work units output by the rasterizer are 2x2 blocks of pixels to operate on combined by the interpolated vertex attributes for the specific location of a pixel within a triangle.

Given the relevance of this component for computer graphics utilizing the traditional raster pipeline, at least one such component can be found on any modern GPU. Current consumer cards can have as many as 12 rasterization units [1].

The amount of rasterizer units and the throughput per unit combine to determine the limit of how many triangles can be processed per unit time.

2.3. Performance Indicators

Even with the most efficient resource usage, there is a physical limits to what specific hardware components are capable of achieving. For understanding the limits of a specific piece of hardware, the

- Clock Rate

  Measured as frequency in Hertz (Hz), i.e. cycles per second, the clock rate determines how frequently transistors switch state. This is a general measurement for how fast data processing is performed. Though comparing clock rate from GPUs of different architectures is generally ill-advised since it does not measure how much work actually is being completed.

  Some components of a system or even within a GPU may run with a different clock rate. Different hardware components may be able to operate properly up to a certain frequency. For instance, it is common for the compute and memory units to operate at different frequencies since compute units are generally able of tolerating higher frequencies than what SDRAM is capable of.

- Memory Bandwidth

  Measured in bits per second, the memory bandwidth of a system puts a limit on how much data can be processed within any given time frame. It is a value derived from the transfer rate of the memory chip combined with the width of the memory bus. The transfer rate gives a measurement for how frequently a data transfer operation can occur and the bus width determines how many bits are transferred per operation. Transfer rate is correlated with clock rate. Double-Data-Rate RAM for instance has a transfer rate roughly twice its clock rate.

[1] see RTX 4090 – Ada architecture [3]

- (Instruction) Latency & Throughput

  Measured in clock cycles, instruction latency is a measurement for how long an operation on a given work item takes from start to completion. Throughput on the other hand measures how long it takes after the completion of one operation for a subsequent operation to also be completed.

  Latency and Throughput are only the same in a serial system. In a parallel system like the GPU, the latency of a single instruction is often hidden by switching to another independent work item and continuing processing there until the results of the first item have arrived.

- Degree of parallelism / Number of cores

  The ability of a GPU to perform processing is tied to the number of available computational units. A single unit capable of working on a single piece of data at a time is referred henceforth referred to as a *core*. The amount of cores on a GPU multiplied by the capabilities of a single core can be used as a measurement for how many items can be processed physically at the same time.

3. GPU Driven Hierarchical Meshlet Rendering

The traditional implementation of a real-time rendering system has the central processing unit (CPU) assigned with the task of generating draw commands and providing data for the GPU to process. Having its origins a co-processor for acceleration for a fixed function pipeline in mind, this was and often still is a reasonable structure to have. However, modern GPUs often have far greater compute capacity available than regular CPUs. When the CPU cannot keep up with providing the GPU with data to process, the GPU can become underutilized. On a desktop computer, data transfers between CPU and GPU usually uses the PCIe connection between the hardware components. The bandwidth available on the PCIe connection is then also a resources that needs to be taken into consideration. The bandwidth of the PCIe connection tends to be much lower than the bandwidth of the GPU's main memory.

In most real time rendering applications, the actual data of the 3D models needed for producing images stays consistent across a large number of frames. Ideally, the scene data would be transferred to the GPU only once and stay there persistently until it is no longer needed. The amount of memory local to a GPU as well as the ability to perform general purpose computations on data means gives the means to do exactly that: Upload all the scene data to the GPU once and have the GPU manage what exactly to draw itself. If a rendering application is structured in such a way that the GPU manages its own work, it is referred to as being *GPU-driven*.

3.1. History of Meshlet Rendering

As mentioned in the introduction, traditional rendering system operate on a per instance granularity. This means for each instance it is determined whether the instance is visible using techniques such as frustum and occlusion culling. If an instance is visible a mesh with a specific level of detail for its representation is chosen. All triangles of this mesh are queued for processing on the GPU. With increasing complexity of 3D models and virtual worlds, the per instance granularity for culling and LOD was found to be too coarse, resulting in large amounts of superfluous work being done and subsequently not being able to achieve the desired visual fidelity.

A solution to handling large amounts of geometry was presented by Ubisoft in Siggraph 2015 [6]. In their rendering pipeline large meshes were broken into individual chunks of a fixed size henceforth referred to as meshlets or triangle clusters.

Breaking a mesh down into meshlets results in a large amount of uniformly sized pieces of data that can be treated mostly independently. This form of data is particularly well suited for the parallel architecture of a modern GPU. In fact, recent generations of graphics cards even provide first class support for a meshlet based rendering in the form of an alternative geometry pipeline utilizing task and mesh shaders [7] instead of traditional vertex, geometry and tesselation shaders. In this alternative pipeline, task shaders are used to spawn a variable number of mesh shaders who then operate on a set of triangles and assembling their own output instead of operating on a single vertex and using the fixed function input assembler. Although certainly useful, task and mesh shaders is not required to take advantages of a meshlet based rendering approach.

3.2. Meshlet LOD hierarchies

The most noteworthy recent innovation in meshlet rendering system comes in the form of Nanite which is part of Epic Game's Unreal Engine 5 and was presented by Karis in Siggraph 2021 [1]. With Nanite geometric complexity of 3D models was able to reach millions of triangles per model and billions of triangles per scene while maintaining both real time interactive performance and high visual fidelity. One major factor in achieving these results was the introduction of a specially formed meshlet LOD hierarchy.

Traditional Level of Detail systems select for a specific 3D model a concrete 3D mesh. Switching between different levels of detail is accomplished by switching between which mesh is rendered. The decision is based on what will from now on be referred to as a *LOD-cut value*. How this LOD-cut value is calculated exactly depends on the application, though it usually involves an error metric based on the distance from the object to the camera and the screen size taken by the object.

To create a LOD of a mesh, i.e. a version with lower amount of detail than some reference mesh, either an algorithmic simplification operation is applied or an artist is tasked with creating a version of the 3D model with using only a constrained amount of polygon / triangles. It is generally desired that a LOD retains as much detail as possible while reducing the amount of geometry used to represent it. However, the any measurement of "detail" only makes sense in the context of a complete 3D model.

Applying a simplification operation on a per meshlet basis results in a simplified version of that meshlet. There are, however, no guarantees that the simplifications done to an individual meshlet make sense in the context of the whole model. This then causes visual cracks or

seams to appear in places where a smooth and continuous surface should be located.

Avoiding cracks between meshlets of different LODs requires that the boundary between two distinct levels of meshlets stays the same. Concretely, the edges and vertices on the boundary between two meshlets are required to be unaffected by any simplification operation. This in turn inhibits the simplification algorithms ability to reduce the number of triangles in the model below a certain amount.

The precise lower bound of triangles per meshlet based on the number of locked vertices can be calculate as:

$$min\_triangles/meshlet = max(num\_locked\_vertices - 2; 1) \tag{3.1}$$

which follows from the fact that any polygon with n sides (n-gon) requires at least $n - 2$ to be triangulated.

In Nanite, the issue with locked boundaries is solved by alternating which meshlets share boundaries on different levels of the LOD-hierarchy. Paraphrasing the process described by Karis, the steps for building such a hierarchy are as follows:

1. **Group**

   Meshlets with a high number shared boundaries are grouped together. The edges and vertices on those boundaries are now unlocked and can affect in the simplification process.

2. **Simplify**

   The meshlet group is now treated as unified collection of triangles. A simplification operation is applied until the number of triangles have been sufficiently reduced (Nanite targets 50 percent). Edges and vertices that are shared with other groups are not affected during the simplification process.

3. **Split**

   The simplified group of triangles is once again split into meshlets of the target size. These new meshlets all receive the same value LOD-cut value based upon the LOD-cut-values of the child and the simplification operation.

4. **Repeat**

   This process can now be repeated until only a single meshlet remains.

3.2.1. Cutting the LOD hierarchy

Building a meshlet LOD-hierarchy creates a directed acyclic graph (DAG). Starting from a root node which represents the most simplified version of a model, the child nodes represent the same surface as their parent meshlets with increasing amounts of detail (see Figure 1).

Selecting a meshlet for an specific region can be done conceptually be traversing the DAG from the root node and stopping once a meshlet has been found that is sufficiently detailed. The selection process used the LOD-cut value of a meshlet in comparison with the LOD-cut value of its parents. However, traversing the DAG the conceptual way is unsuited for a parallel architecture since it creates a long, serial dependency chain starting from the root node down to the leaves of the DAG. Instead, there needs to be a way to determine the position of the cut on a per meshlet basis.

For this to be possible, it is required that the LOD-cut value is monotonically increasing or decreasing within the DAG. Without such a monotonic function, the decision process is ambiguous and may result in meshlets being drawn over each other. Ensuring the correct relationship between values can be done in a post-processing step of building the hierarchy. Alternatively, it must be ensured that the function used to create the LOD-cut value of a simplified group of parent meshlets is always greater (or always less) than the maximum (or minimum) value of the children from which they were created[1].

Ensuring that the LOD-cut value is monotonic within the DAG is mathematically simple. The challenge lies in keeping the LOD-cut value useful for choosing between different levels of detail based on view distance and screen size. What exactly the LOD-cut value represents is application specific, though there are two general orientations that would give a useful metric:

- Visual Error Oriented

  A LOD-cut value based on the same or similar metric as used by the simplification processes to determine which triangles to remove. Such a metric measures how similar or divergent two representations of an object are. In informal terms, this type of metric decides for a meshlet whether it is "close enough" or "good enough" to not cause more than an accepted amount of visual difference.

- Execution Efficiency Oriented

  A LOD-cut value based on the density of triangles on screen. As will be analyzed in a later chapter(4), the size of a rasterized triangle has a noticeable impact on how efficiently pixels generated by this triangle can be processed. A metric for determining how efficiently a meshlet will on average be processed is therefore a useful measure for selecting a level of detail although potentially at the cost of decreased visual fidelity.

[1] The parent-child relationship in this DAG is biologically as inverted as the growth direction of trees.

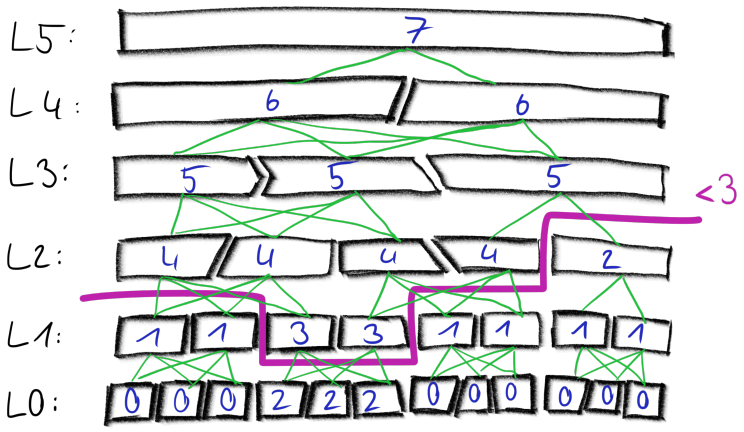


Figure 1.: An example LOD-hierarchy, Box number is LOD cut value, green lines represent parent-child relationships, meshlets under the red line are chosen for for a cut at LOD-cut value < 3

In practice, these two directions for LOD-cut metrics span a spectrum. The approach taken in Nanite while primarily visual error based has noticed a significant improvement in execution consistency once error was calculated relative to the same meshlet size for every meshlet.

3.2.2. Incremental LOD-Hierarchy

Building the meshlet LOD-hierarchy level by level decreasing the amount of triangles each step by a consistent amount is beneficial for an LOD-cut metric based on visual error since the error will roughly uniformly increase across the mesh. The resulting DAG is very uniform, with each level containing a roughly fixed fraction of the number of meshlet of the previous level. For primarily execution efficiency oriented LOD-cut metrics on the other hand, having a such uniformity in the DAG might be undesirable. Meshes where the triangle density varies strongly throughout the mesh, have a high chance for initially large meshlets to propagate their density across the mesh causing small details to be removed earlier than necessary.

As far as research for this thesis could determine, this is a yet to be addressed issue when building a hierarchical meshlet rendering system [2]. This thesis' solution proposal is simple:

[2] In practice, it might not manifest as an issue, especially if LOD levels are very wide or meshlets mostly evenly sized

Build the LOD hierarchy incrementally, by applying grouping, simplify and split operations only to regions with high density meshlets. Meshlets that are less dense are virtually moved up the hierarchy until it makes sense for them to be simplified as well.

3.3. GPU Draw Command generation

With the meshlet hierarchy in place, the GPU-driven rendering pipeline now needs to determine which meshlets to draw and which to ignore. Concretely, the GPU needs a way to conditionally queue a draw command that will be scheduled for processing.

Even in GPU-driven rendering, queuing commands for the GPU to execute is done by recording a command buffer on the CPU side and sending it over to the GPU's command decoder. Common graphics APIs, currently provide no means to submit a command directly from a shader program. What is provided instead is a family of commands where part of the command is to fetch the arguments from a specific memory buffer. These are called *indirect* commands since they provide their parameters indirectly.

For example. the Vulkan specification [8] defines the following structure of one such command parameters entry:

```
typedef struct VkDrawIndexedIndirectCommand {
    uint32_t indexCount;
    uint32_t instanceCount;
    uint32_t firstIndex;
    int32_t vertexOffset;
    uint32_t firstInstance;
} VkDrawIndexedIndirectCommand;
```

Writing to a buffer containing such a command parameters structure, allows the GPU to set both how many triangles are drawn (indexCount) as well as how many instances of the mesh are drawn (instanceCount and firstInstance). It can also select a sub-range inside a vertex and index buffer (firstIndex, firstInstance).

With Vulkan, DirectX12 [9] and Metal [10] , there is additionally a way to dispatch compute shader via an indirect command. For Vulkan, this command consists of the following parameters:

```
typedef struct VkDispatchIndirectCommand {
    uint32_t x;
    uint32_t y;
    uint32_t z;
} VkDispatchIndirectCommand;
```

The dispatch indirect command is much more limited in terms of functionality. Only the number of work groups in each dimension can be adjusted.

3.3.1. Dynamic Index Buffer Generation

Indirect commands enable the GPU to specify in the contents of a single command. However, the CPU still needs to declare how many such indirect commands should be queued for execution. An issue presents itself in a GPU-driven pipeline: The information how many commands need to be queued is not available until the GPU has selected what to draw and what to exclude. Communicating this information back is typically not a viable solution as it would require synchronizing CPU and GPU execution and will likely go through a high latency channel such as the PCIe connection.

One option to avoid communication would be to prepare for the worst case scenario and dispatch as many commands as potentially required, setting the actual amount of work in a command to zero if nothing should be executed. Unfortunately, command processing is also a form of work that the GPU needs to execute. Processing a command that leads to no result is by definition inefficient.

An alternative solution is presented by Ubisoft [6]: In their implementation, only a single indirect draw command is executed per view / virtual camera and the index buffer is dynamically generated each frame to include the triangles that need to be drawn. The index count variable of the indirect draw command is then set to render as many triangles as were appended to the dynamic index buffer. This approach has the additional benefit that it can even work on a per triangle granularity. However, the amount of data that is transferred is far from ideal. The memory bandwidth requirements scales linearly with the number of triangles per visible meshlet. For one triangle. three 32 bit integers need to be written. For a meshlet with 64 triangles that means that $32bit \times 3 \times 64 = 6144bit = 768byte$ must be copied. For contrast, a draw command is only $32bit \times 5 = 160bit = 20byte$. If it would be necessary to use meshlets with higher triangle counts, for example to better saturate a cluster of SIMD-compute units, it may lead to a point where processing an empty command may incur less overhead than transferring this much data per visible meshlet. Ideally, the required memory bandwidth for visible meshlets should only scale with their actual number.

3.3.2. Multi Draw Indirect Count

Multi-draw-indirect is a special indirect command that allows the CPU to specify a batch size of > 1 for indirect commands to be queued for execution in a single command sent to the GPU. Building upon this, Multi-Draw-Indirect-Count is a commonly available extension to this feature allowing to fetch the batch size from a buffer, too. This gives an effective way for

the GPU to schedule an arbitrary amount of draw commands entirely on its own. Emitting one small draw command parameter structure per meshlet saves a significant amount of memory bandwidth. Clearly, this approach should be preferable to the previous options if it is available.

4. Triangle Rasterization Efficiency

4.1. Software vs. Hardware Rasterizer

One of Nanite's defining features is the implementation of a software rasterizer. It is claimed that for small triangles, their software rasterizer is able to be up to three times faster than the hardware rasterizer. The impression is given that the rasterizer units of modern GPUs are not fast enough setting only up to 4 triangles per clock cycle. Such claims warrant further investigation, especially since it would imply major changes could be made to future GPU designs to address lacking rasterizer performance.

Note: Since to "setup a triangle" is not a clearly defined operation, for the sake of further analysis the more concrete operation of emitting a single 2x2 pixel quad for fragment shading work will be assumed to be equivalent.

Consulting architecture references from both Nvidia and AMD, the rasterization units in their respective GPUs are indeed stated to be able to process up to one triangle per cycle per rasterization unit. The amount of rasterization units therefore correlates with the amount of triangles that can be processed per cycle. GPU with four rasterization units would consequently only be able to setup a maximum of four triangles per cycle.

For Nanite's rasterization stages, $1148s + 183us$ for $24,601,344$ software rasterized triangles total were reported. This gives a rasterization rate of $18.5 \times 10^9 triangles/second$. These times where measured on a Sony PlayStation 5 which has its GPU core clocked @2233 Mhz. Assuming four rasterization units to be present[1], the maximum expected throughput of the hardware rasterizer units is with $8.9 \times 10^9 triangles/second$, indeed a over two times slower than Nanite's software rasterizer.

It must be stated that these back of the envelop calculation assumes that every triangle actually produces at least a single output pixel (or 2x2 pixel quad in case of the hardware rasterizer). Triangles that are not being rendered, for example because they are facing in the wrong direction (backface culling) or are discarded early because they do not cover any

[1] The GPU is based on the RDNA 2 architecture for which the number of rasterizer units is unknown. However, the immediate predecessor architecture had around one rasterization unit for every ten compute units which is a ratio that is unlikely to have changed drastically

pixel centers are not factored into the assumptions. However, even assuming the triangle throughput of the software and hardware rasterizers were identical, for small triangles, the software rasterizer would still provide an advantage. The reason for this will be explained in the following sections.

4.2. Helper Invocations

When triangles are converted into 2x2 pixel quads by a hardware rasterizer, not every pixel in those quads is necessarily also inside of the triangle. These outside-pixel result in so called *Helper Invocations*. Helper invocations execute the same instructions as regular invocations but do not directly contribute their result to the final image. They are, however, needed for finite difference derivative calculations (e.g. required for mip-map level selection).

Despite their necessity, helper invocations do require compute resources for little to no contribution to the final result. This makes them a source of inefficiency, so they should be avoided as best as possible. Fortunately, the occurrence of helper invocations is very predictable as can be verified experimentally.

4.3. Theoretical Analysis

Let 2^n with $n >= 1$ be the width and height of a pixel grid cell which is spanned by two right triangles. The number of 2x2 pixel quads in the grid cell is then given by $2^n/2 = 2^{n-1}$ for each row, column and diagonal respectively. The total number of 2x2 quads per cell is $2^{n-1} * 2^{n-1} = 2^{2*(n-1)}$.

Each quad on the diagonal is processed twice, once for triangle that span the grid cell. This increases the effective amount of processed quads to $2^{2*(n-1)} + 2^{n-1}$.

The increase processing cost can now be formulated as a percentage value of the grid dimensions given as $(2^{2*(n-1)} + 2^{n-1})/2^{2*(n-1)}$ or more concisely: $(2^{n-1} + 1)/2^{n-1}$.

4.4. Experimental Examination

4.4.1. Preparation

A frame buffer with square dimensions is subdivided into a grid of rectangles. Each rectangle in turn is formed by of two triangles. The frame buffer is at least double buffered or triple buffered if the driver requires it. There is no depth test performed and the frame buffer images are not cleared between draw calls. Results are presented to screen but vertical sync is turned off to not pollute processing time with idle periods.

Vertex data is packed into a linear buffer and consists of two 32 bit floating point values for x and y clip space coordinates. An index buffer with 32 bit integers is used to assemble the triangles.

The vertex shader performs a simple pass through of the vertex data. No additional computations or memory fetches are introduced.

The fragment shader performs a calculation which requires a large amount of compute cycles to complete. This calculation is based on the current fragment position to avoid the shader compiler pre-calculating the result. Specifically, this is done by introducing a long serial chain of sinus computations. Additionally, the partial derivatives in x and y of the computation result are taken. These three obtained values are then stored in the frame buffer as r,g,b color values.

For submitting work to the GPU, command buffers are recorded once for each image of the frame buffer once the image becomes available for the first time. After the first usage of an image, the command buffer is resubmitted when the image becomes available again. Combined with the double or triple buffered structure of the frame buffer, this should be sufficient to keep the GPU saturated with work.

The benchmark code is written in C++ and utilizes the Vulkan Graphics API. Shader Code is written in GLSL and compiled to SPIR-V. The GPU used for testing is an Nvidia MX250 mobile GPU. The exact environment should have no influence on the conclusion of the experiment as every GPU currently on the market should exhibit the same behaviour and any graphics API can be used to construct a similar experiment setup. [2]

4.4.2. Execution

1. A base line measurement with zero helper invocations is taken using a single triangle that covers the entire frame buffer.

2. Per subdivision level – ranging from 1 grid cell per frame buffer to to 4 grid cells per pixels – vertex and index buffers are created.

3. For each subdivision, the time between acquiring the first frame buffer image and the completion of a specified number of frames[3] is taken. The total execution time is averaged over the number of rendered frames to receive the average time of completion per frame in micro seconds.

4. The process was repeated with a higher resolution frame buffer for verifying the scaling behaviour.

[2] Similar experiments g-truc and SimonDev though they do not provide exact numbers

[3] 2048 frames for the baseline measurement, 1024 for the subdivisions. This is precise enough given that even when rendering at 1000 frames per second, the interval between time measurements is greater than a second

4.5. Evaluation

grid cell size	run time µs	measured cost	predicted by formula
Baseline	8062,6	1,0	1,0
1024	8078,8	1.002009277	1.001953125
512	8115,8	1.006598368	1.00390625
256	8172,3	1.013606033	1.0078125
128	8177,9	1.014300598	1.015625
64	8288,5	1.028018257	1.03125
32	8468	1.050281547	1.0625
16	8880,5	1.101443703	1.125
8	9791,2	1.214397341	1.25
4	11967,9	1.484372287	1.5
2	16190,8	2.008136333	2
1	31653,2	3.925929601	(3)
0,5	28035	3.477166175	X
0,25	30870	3.82878972	X

Table 1.: Raw data for Nvidia MX250, 1024x1024 frame buffer resolution.

Comparing the predicted and the measured execution times shows and as is visualized in figure 2, the theoretical increase in rendering costs is nearly identical to the measurements up until a grid size of 1 pixel.

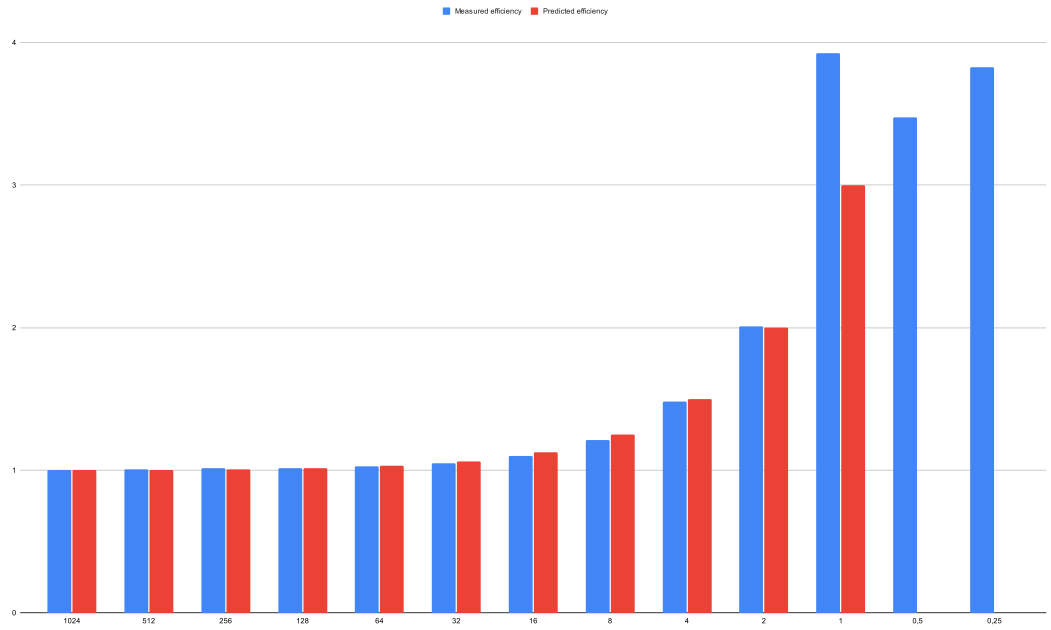
Below this size, only a one of the triangles spanning the grid results in a fragment shader invocation, the other one is not covering a pixel center anymore and is thus discarded. The invocations that do occur, however, operators at only 25% efficiency because an entire 2x2 quad / 4 pixel need to be processed even if only a single pixel will be displayed.

Measurements for sub-pixel grid sizes are a bit noisy but generally stay at around the same frame time. Although some fluctuations are to be expected since triangles have become so small that some are not producing any fragment shader invocations and while still requiring some processing in the rasterizer unit.

Hardware specific:

Depending the number of units and the clock speed of the GPU, the hardware rasterizer may or may not become an actually measurable factor in this experiment. For the 1x1 grid cell size on a 1024x1024 frame buffer, around 2 million triangles are processed each frame. A (mobile) GPU with a less optimized rasterizer, with an assumed throughput of 10 cycles per triangle clocked @500 MHz, would only be capable of processing 1.6 million triangles

Figure 2.: Measured vs. theoretical performance cost increase based on triangle size (Nvidia MX250)



while still achieving a 30 fps target.

On the other hand, a higher clocked GPU and/or a GPU with a better rasterization throughput of 1 triangle / cycle will not be significantly affected. On such a GPU, a single rasterization unit at a clock speed of 1500 MHz [4], between up to 25 million triangles per frame could theoretically be achievable while still reaching a 60 fps frame rate target. For this experiment is is therefore unlikely that such a GPU would show a significant bottleneck through the hardware rasterizer.

4.6. Conclusion

Small triangles invoke proportionally more helper invocations than larger triangles. This can result in an up to 3x higher fragment shading cost per triangle and over 10% increase in fragment shading cost is measurable as early as triangles covering 8 pixels (16x16 grid cell).

[4] this is clock speed of the Nvidia MX250 laptop GPU

Special care needs to be applied when dealing with small triangles, to avoid an excessive amount of helper invocations.

In Nanite, triangles with an edge length below 32 pixel are handled by a software rasterizer. Up until this size, Karis reports that their implementation is superior to the hardware path. While no numbers are provided for this claim specifically, given the processing cost increase for this triangle size, even a software rasterizer only on par with the hardware rasterizer would see a difference of 5-6% in processing time. The claim of the software rasterizer being up to 3 times faster would then correspond to saving the 3x cost increase measured for triangles below the 2x2 block size. This makes software rasterization an attractive option for handling very small triangles.

The next best practical solution if building an optimized software rasterizer is not an option, is to avoid small triangles as best as possible. If no small triangles are submitted for processing, resources in the rasterizer are utilized better. Using larger triangles may result in lost detail and decreased visual fidelity although this can be compensated for with established techniques such as normal or parallax mapping. When limited to a small area, the difference between such techniques and actual geometry can be made close to being imperceptible. However, artist would now be tasked to create the necessary resources required for these techniques (normal maps, height maps, etc) which may increase development costs of the application.

A third solution would be to render at a lower resolution and rely on image up-scaling techniques such as DLSS [11], FSR [12] and X^{e}SS [13] for a good mix between visual quality and execution time. Halving the screen resolution in each dimension reduces the number of pixels by 4 times. This would compensate for the 3x increase in effective fragment shader cost. If triangles are at least quad sized and the cost increases only by a factor of 2x, the render target resolution would only need to be reduced by 25% for compensation.

5. Rendering Pipelines

When building an application for interactive computer graphics, a choice must be made on how to structure the pipeline of geometry being transformed from their triangle representation to colors on screen. While GPUs present a predetermined path for this process, applications are free to choose how to walk this path. Depending on the needs of the application, using the processing pipeline of the GPU in different ways can facility an overall better resource usage. It thus needs to be evaluated which implementation of a rendering pipeline would best suit the application.

5.1. Forward Rendering

5.1.1. Standard Forward

Standard forward is the expression of the classic rendering pipeline and the pipeline that follows the GPU's processing pipeline the closest. In simple terms, the pipeline for a single triangle starts with calculating whether it is located within the bounds of the screen (or render target) and if so, its color is calculated for every pixel it covers. More concretely, the steps for each work item in the application are as follows:

- For each vertex:

  The vertex position and other attributes are fetched from memory. Both position and attributes are then transformed to screen space.

- For each triangle:

  The transformed vertex data is passed to the rasterizer where attributes are interpolated and 2x2 pixel quads are emitted per screen coverage.

- For each emitted 2x2 pixel quad:

  Additional material data is fetched from buffers and textures and the lighting calculation is performed.

This gives the following equations for estimating the hardware requirements to perform standard forward rendering given a specified amount of work items to process:

Memory bandwidth:

$$\begin{aligned} & 1 \times vertex\_index \times num\_vertices \\ & + 1 \times vertex\_position \times num\_vertices \\ & + 1 \times vertex\_attributes \times num\_vertices \\ & + 1 \times instance\_transformation\_data \times num\_vertices \\ & + 4 \times material\_data \times num\_emitted\_quads \end{aligned} \tag{5.1}$$

Compute work:

$$\begin{aligned} & 1 \times vertex\_position\_caluclation \times num\_vertices \\ & + 1 \times attribute\_transformation \times num\_vertices \\ & + 4 \times lighting\_calculation \times num\_emitted\_quads \end{aligned} \tag{5.2}$$

The major driver for resources demands is the number of emitted 2x2 pixel quads. If two or more triangles cause quads to be emitted for the same area of an image, the computation for all but one of those quads is wasted work. Overwriting work of a previous calculation in computer graphics is commonly referred to as *overdraw*. The amount of overdraw is inversely proportional to the efficiency of the rendering processes with less overdraw equates to less wasted work and thus higher efficiency. Standard forward rendering is most efficient when the amount of overdraw small. Then the amount of processing work done is driven primarily by what the application requests to process.

5.1.2. Forward + Depth Pre-Pass

In practice, a certain amount of overdraw is often unavoidable. Although when overdraw occurs, it has no influence on the appearance of the image. This is because when a pixel's color is attempted to be written, a depth value for this pixel is emitted as well. This depth value is then stored in a depth buffer. When attempts are made to write to the same pixel again, the depth value stored in the depth buffer is used to decide which of the results to keep and which to discard. The contents of the pixel color buffer and the depth buffer are then updated accordingly. In the classic pipeline, this is done after the calculation has taken place.

The depth value of a pixel itself tends to be solely determined by the position of the source triangle. In such a case, it can be checked whether the color calculation even has a chance to change the result. In modern GPUs, this is implement as an optimization called the *Early-Z*-test. Early-Z performs the depth test for a pixel/fragment before the corresponding fragments shader is actually executed. This test is typically part of the rasterizer unit's internal

pipeline. So if all pixel of a 2x2 quad fail the early-Z test, it does not get emitted for further processing thus saving computation time.

Advanced forms of forward rendering (Forward+) take advantage of the early-Z test feature by processing geometry in two passes. The first pass is used to only fill in values in the depth buffer. The second pass then re-evaluates all geometry but performs the actual calculations only where the depth buffer values match. This changes the equations for the cost of forward rendering to the following:

Memory bandwidth:

$$\begin{aligned} &2 \times vertex\_index \times num\_vertices \\ &+ 2 \times vertex\_position \times num\_vertices \\ &+ 1 \times vertex\_attributes \times num\_vertices \\ &+ 2 \times instance\_transformation\_data \times num\_vertices \\ &+ 4 \times material\_data \times num\_second\_pass\_quads \\ &+ 1 \times (Read + Write) \times depth\_buffer\_value \times num\_emitted\_quads \end{aligned} \tag{5.3}$$

Compute work:

$$\begin{aligned} &2 \times vertex\_position\_caluclation \times num\_vertices \\ &+ 1 \times attribute\_transformation \times num\_vertices \\ &+ 4 \times lighting\_calculation \times num\_second\_pass\_quads \end{aligned} \tag{5.4}$$

The number of quads emitted in the second pass should allow for a close to one-to-one ratio of lighting calculations done per pixel. The issue of overdraw in this pass is solved by the early-Z test, saving potentially a large amount of fragment shading work. However, this comes at the cost of additional vertex processing being performed and increased demands on the memory system since all vertex positions need to be processed twice. Doing two passes of geometry also increases demands on the rasterizer since assembling and processing triangles that are ultimately discarded also requires processing time. Processing every triangle twice constitutes a low efficiency in the second geometry pass as proportional to the amount of overdraw triangles are now expected to be discarded.

Hardware specific:

Every GPU implementing the early-Z optimization can potentially benefit from a depth pre-pass. Without such an optimization, calculations would still be performed before being discarded. The amount of computation work being saved must also outweigh the increased

memory bandwidth required for processing all geometry twice. On mobile GPUs, doing a depth pre-pass is discouraged for this reason.

5.2. Deferred Rendering

Deferred Rendering [14] is a technique introduced to reduce the impact of overdraw on the computation units by avoiding expensive lighting calculations. This is done by deferred the actual calculation until every piece of geometry has been evaluated. In place of computing the light, only the data later needed for this calculation is stored in a so called G-Buffer. A second pass over all pixel fetches this data to perform the actual calculation. This gives the following breakdown of resource demands:

Memory bandwidth:

$$\begin{aligned} & 1 \times vertex\_index \times num\_vertices \\ & + 1 \times vertex\_position \times num\_vertices \\ & + 1 \times vertex\_attributes \times num\_vertices \\ & + 1 \times instance\_transformation\_data \times num\_vertices \\ & + 1 \times (write) \times G\_buffer\_data \times num\_emitted\_quads \\ & + 1 \times (read) \times G\_buffer\_data \times num\_pixel \\ & + 1 \times material\_data \times num\_pixel \end{aligned} \tag{5.5}$$

Compute work:

$$\begin{aligned} & 1 \times vertex\_position\_caluclation \times num\_vertices \\ & + 1 \times attribute\_transformation \times num\_vertices \\ & + 1 \times lighting\_calculation \times num\_pixel \end{aligned} \tag{5.6}$$

Most notably, deferred rendering achieves an ideal amount of lighting calculations per pixel. When the amount of overdraw and the cost of fragment shading is high, this will save a significant amount of computation. However, the trade off is a significant increase in required memory bandwidth. The lighting calculation requires a large amount of input data such as position, normal and tangent vectors, surface color and surface parameters. Since this data needs to be stored in the G-Buffer, the G-Buffer naturally requires a large amount of storage space. Although G-Buffer data can be stored in a compressed format, the minimum amount of bits per pixel is still comparatively high ($> 160 bit/pixel$) [15].

Hardware specific:

Deferred rendering approaches require the ability to read and write reasonable amount of user-defined data to textures. A sufficiently high memory bandwidth also needs to be available for this approach to be viable. The bandwidth requirements also scale with render resolution instead of with processed geometry. Cases where processing all geometry twice requires less bandwidth than storing and loading G-Buffer data make a Forward+ pipeline preferable over deferred rendering.

On tile based mobile GPUs, the G-Buffer data for deferred rendering can often be stored and accumulated in tile-memory, provided that there is a 1:1 mapping between accesses to a G-Buffer item and pixel performing lighting calculations. The contents of the G-Buffer then do not need to be written to main memory, thus, avoiding expensive data transfers. However, the amount of data tile memory can hold per pixel may be less than what the G-Buffer would require.

5.3. Visibility Buffer Rendering

The concept of a visibility buffer was introduced as a specialized approach of deferred rendering which makes better use of the GPU's caches [16]. As the name implies, a visibility buffer only stores which triangle is visible for any given pixel. As only a depth value and an identifier are stored, the memory footprint of the buffer is minimal.

Same as with regular deferred rendering, all geometry is processed once fill the visibility buffer. In a second pass over all pixel, the triangle identifier is then used to fetch all data from the vertex attribute buffer, interpolate them for the current pixel position and finally perform the lighting calculation. This approach gives the following equation resource demands:

Memory bandwidth:

$$\begin{aligned} & 1 \times vertex\_index \times num\_vertice \\ & + 1 \times vertex\_position \times num\_vertices \\ & + 1 \times instance\_transformation\_data \times num\_vertices \\ & + 1 \times (write) \times triangleID \times num\_emitted\_quads \\ & + 1 \times (write) \times depth\_buffer\_value \times num\_emitted\_quads \\ & + 3 \times vertex\_index \times num\_pixel \\ & + 3 \times vertex\_position \times num\_pixel \\ & + 3 \times vertex\_attributes \times num\_pixel \\ & + 1 \times instance\_transformation\_data \times num\_pixel \\ & + 1 \times (read) \times triangleID \times num\_pixel \\ & + 1 \times material\_data \times num\_pixel \end{aligned} \tag{5.7}$$

Compute work:

$$\begin{aligned} & 1 \times vertex\_position\_caluclation \times num\_vertices \\ & + 1 \times position\_reconstruction \times num\_pixel \\ & + 3 \times attribute\_transformation \times num\_pixel \\ & + 1 \times attribute\_interpolation(software) \times num\_pixel \\ & + 1 \times lighting\_calculation \times num\_pixel \end{aligned} \tag{5.8}$$

As can be seen, almost every resource demand scales primarily with the number of pixel in the visibility buffer. As with standard deferred rendering, an ideal ratio of lighting calculations per pixel is achieved while at the same time requiring significantly less memory bandwidth and intermediate storage space. When memory caches are able to avoid duplicate requests for the same pieces of data, vertex attributes are fetched only once per visible triangles instead of for all triangles, thus, potentially saving additional bandwidth. The trade-off, however, is previously hardware accelerated attribute interpolation cannot be used anymore and must instead be performed by the general purpose compute units.

Hardware specific: A visibility buffer approach requires the most flexibility in terms memory access permissions. Crucially, the contents of the index buffer and the vertex attribute buffer must be readable with a random access pattern in fragment shader. GPUs may have optimized forms of storage for these buffer types, a feature that cannot be taken advantage of when using a visibility buffer. Substituting hardware accelerated processing paths with

general purpose compute to work on fewer work items in total might result in more processing time than working on more items in an accelerated way.

Though as shown with Nanite's software rasterizer, the hardware fixed function path might be outperformed by a software implementation. Using a software rasterizer is only made viable for a visibility buffer rendering pipeline. The data size per buffer entry is low enough that atomic operations can be used to modify the buffer's content.

Since the only operation needed to fill the visibility buffer is writing a single value per pixel, a software rasterizer can be optimized for queuing memory write requests as efficiently as possible. The hardware rasterization unit on the other hand would still output 2x2 pixel quads, some of which may not result in a write request due to being a helper invocation.

5.3.1. Triangle/Primitive ID

Visibility buffer implementations without a software rasterizer also requires that a triangle identifier can be passed through the rasterizer to the fragment shader. Unfortunately, the conversion step in the rasterizer normally discards or never obtains the information which pixel belongs to which triangle. The data path for vertex attributes must then be used to relay this information. However, this in turn requires that for any given vertex it is known which triangle it belongs to. In the general case, this is unambiguous since the same vertex is likely to be shared by multiple triangles. Ideally, the GPU would be aware of which triangle is currently being processed and would provide a way to obtain this information.

GPUs that support mesh shaders have the most direct support for providing triangle identifiers since assembling the input vertices into triangles is done in a user defined way. However, this shader pipeline type is currently still reserved for recent desktop GPU. Fortunately, more widely supported shader types such as geometry and tesselation shaders require triangles to be identifiable by a sequence number. The presence of either of these shader types, thus, correlates with the hardware's support for exposing a triangle primitive ID. The Vulkan specification for example states that such a primitive ID can only be provided if either of these features is enabled.

When no primitive ID can be obtained directly from the GPU, alternative methods to identify a triangle from the processed vertices need to be employed. One solution is to *unroll* the mesh, i.e. forego the index buffer and have every triangle as a set of three individual vertices in a large vertex buffer. The triangle ID is then given by $\lfloor vertex\_index/3 \rfloor$.

This, however, is not ideal for various reasons. One of which is the increase in memory storage and memory bandwidth demand for processing vertices. An additional downside is that a special cache called the *post transform* cache can no longer be used. For draw commands using an index buffer the results of a vertex transformations are stored and retrieved should another triangle contain a previously processed vertex. Since vertices are

no longer shared in an unrolled mesh, duplicate processing needs to be done. This may outweigh the savings enabled by the visibility buffer pipeline.

5.3.2. Provoking Vertex: Triangle ID for Meshlets

Every vertex needs to provide information to identify the triangle it belongs to which could be done with a special vertex attribute. However, adding a triangle ID to the attribute of every vertex would make every vertex unique. Unique vertices are not shared so the result would be the same as unrolling the mesh. It is also inefficient to store the triangle ID per vertex as only a single copy of this data would suffice but it is actually present trice. Ideally, only one vertex would be used to store the information to identify one unique triangle. Vertices identifying other triangles could then still be shared with other triangles,

It is possible to disable attribute interpolation performed by the rasterizer. This is referred to as *flat shading*. The value for all pixel covered by a triangle will then be determined by only one of the vertices. The vertex used for this is called the *provoking vertex*. Which of the vertices that compose a triangle will be used as the provoking vertex is well defined by the graphics APIs. Generally, either the vertex with the lowest or the highest vertex index is chosen. The order is also defined when using an index buffer, in which case the order of the indices is used to determine which vertex is considered first or last. Using this insight, the provoking vertex can be used to uniquely identify a triangle while still maintaining a high degree of vertex sharing. It only needs to be ensured that the vertex for identifying a triangle is always placed at the same relative position within the index buffer.

For meshlets rendering, the relative position of a triangle within a meshlet is important, but vertices are shared across meshlets. Computing an assignment of vertex to local meshlet identifier which works for all meshlets of a mesh is difficult if not sometimes impossible. Computing such an assignment per vertex on the other hand is fairly straight forward since there will necessarily be at least one unassigned vertex per unaccounted triangle. A simple solution thus presents itself: Create a unique copy only for vertices on the border of meshlets. The amount vertex duplication should then be at a minimum and using an index buffer is once again possible.

5.4. Takeaway

Use cases for meshlet rendering arrive when the geometric complexity is high and overdraw is likely to be an issue. Such a situation can take advantage of the benefits provided by a visibility buffer rendering approach. However, for rendering of low complexity scenes with little overdraw or for rendering pipelines where additional counter measures against overdraw are implemented, an improved forward rendering approach might be preferable.

The presence of hardware support for features required by a visibility buffer approach also need to be taken into consideration. Some hardware may have specialized data paths which cannot be used with visibility buffer. Not using the specialized data path may lead to inferior performance even when overall less work items are processed. In such a case, even regular deferred rendering might be preferable over the visibility buffer approach.

Designing a GPU architecture to provide first class support for a visibility buffer pipeline would be interesting to explore. However, this is beyond the scope of this thesis.

6. Mesh Cluster Generation

Any meshlet based rendering pipeline needs a way to split a large 3D mesh into meshlets. In hierarchical meshlet rendering, this creates the first level of the hierarchy from which all other levels are constructed. Poor choices in construction on this basic levels can reduce the quality of the hierarchy as a whole, so it is necessary to ensure that these first meshlets have certain characteristics:

- **Triangles within a meshlet should be connected**

  This is required to minimize the error introduced when applying simplification operations. Triangles will not be connected if their vertex positions or vertex attributes do not match. This sometimes is deliberate, for example to create distinct features in the 3D model such as sharp angles, borders between different components, etc.

  Attributes from unrelated triangles often also cannot be linearly interpolated without giving a nonsensical result. Texture coordinates for example may look up totally unrelated data when a point between two random locations within the texture is sampled. Unrelated triangles can thus not be merged for simplification without introducing visual artifacts.

  Connected triangles on the other hand already transitively blend smoothly between vertex attributes. The error introduced by removing vertices between transitively connected vertices only results in a "less smooth" blend between the attributes. This is an acceptable error whose impact on visual quality can be quantified.

- **Borders between meshlets should be minimal**

  When constructing the hierarchy, vertices on the border between two meshlet groups must not be affected when applying the simplification operation, otherwise LOD-cracks will be introduced. More vertices being locked means fewer vertices and edges are available to reduce the meshlet group triangle count.

  When the triangle count of a meshlet group cannot be reduced to the desired amount, more meshlets need to be added to the group until it is possible. More meshlets per simplification group result in coarser control over LOD boundaries. There is then an increased chance that a less optimal meshlet needs to be chosen in LOD-selection which can increase processing costs or reduce visual fidelity.

- **The number of triangles per meshlet should be a multiple of the GPU batch size**

  The GPU's SIMD units have a number of lanes for processing a batch of work in lock-step parallel. The size of a batch should be chosen such that no lanes remain unused or are inactive. A meshlet submits a batch of triangles to process which will then be scheduled on the SIMD units. Empty space in the batch means more work batches need to be processed for the same amount of triangles. This reduces the overall amount of triangles that can be processed per unit amount of time.

  Meshlets with fewer triangles than the target size also cause issues by introducing more padding triangles or introduce complexity in handling varying meshlet sizes. More padding triangles require more memory storage and bandwidth and or introduce additional computational overhead when preparing meshlets for rendering.

Given these characteristics, the importance to ensure a high quality when preparing the meshlets is clear. The quality of an algorithm for meshlet generation can be evaluated based on how well it satisfy or achieves the stated criteria. Unfortunately, finding a grouping into meshlets where all three points are maximally satisfied is an NP-complete problem.

6.1. Graph Partitioning

The task of finding a good division of a mesh into meshlets is a graph partitioning problem. Specifically, it is the problem to find a balanced partition (meshlets all have the same target size they should achieve) with a *minimum k-section* width (minimal boundary length between meshlets) of a graph. The *graph nodes* are the triangles and the *graph links* are the connecting edges between the triangles[1]. A cut is then a selection of graph-links that represent the (shared) boundaries between meshlets.

6.1.1. Minimum k-cut

The complexity of finding a minimum width k-cut in a graph, i.e. a cut that separates a graph into k-connected components with the least amount of links between components, is polynomial if k is fixed [17]. The optimal algorithm's complexity class is in O(n^{k^2}) where n is the number of nodes in the graph. However, the number of partitions / number of meshlets, k, is not fixed in our case. Instead, it is based on the input size given as $k = \lceil n/target\ meshlet\ size \rceil$. This means our version of the problem has exponential time

[1] This is the so called dual-graph of the vertex graph defined by the 3D mesh. Names for graph components "nodes and links" chosen more commonly used "vertices and edges" to avoid confusion with similarly named vertices and edges of the 3D mesh

complexity of O(n^{n^2}). Worse yet, even if in practice the number of partitions were to result in a manageable complexity, a minimum k-cut does not guarantee even partition sizes, thus, nor manage to satisfy the third requirement.

6.1.2. Minimum width bisection

Fixing k to be $k = 2$ and focusing instead on making the partitions balanced describes the *minimum width bisection* problem. The task here is to find a set of graph links with minimal set-size that separate the graph nodes into two connected components of equal size. This problem, too, is NP-complete [18].

The number of distinct sets containing a specific amount of graph-links grows by the binomial coefficient of $\binom{L}{s}$ *s* is the set size and *L* is the number of distinct graph-links (connections between triangles). For a graph with 100 links and a known minimum-bisection-cut set size of 10 the number of possible sets is $\binom{100}{10} = 1.7 * 10^{13}$. While this amount might still be manageable with modern computer systems, the expected amount of links between triangles $> 250,000$ [2]. Assuming a relatively short minimum path of 20, the number of sets to consider is already $\binom{250.000}{20} = 3.7 * 10^{89}$, even higher for longer paths.

Although it is easy to verify whether a given solution has found a valid bisection – count the number of nodes on either side – verifying that this solution is of minimum width can only be done by demonstrating that no smaller set can exists which also would creates a valid bisection. While some approximate lower bounds can be found using for example an algorithm to find the width of a minimum cut, the number of possible sets check even for this lower bound quickly eclipses the estimated number of atoms in the observable universe. Consequently, a brute force search is impractical and the use of approximations and heuristics to find a good solution within some error-bounds is required.

6.1.3. Multi Level Partitioning

Nanite's solution is built upon the METIS library, which implements a multi level partitioning scheme to graph partitioning problems [19].

The general idea for this type of approach is that a solution can quickly be found in a small graph. When the small graph is similar to a larger graph, the solution found there can be used to find a good solution in the large, original graph, too. To find a small but similar version of a graph, the original graph is coarsened by combined nodes while retaining information that later enables back-projecting. Combining nodes needs to employ heuristics and as such does not guarantee that an optimal solution in the coarsened graph leads to an optimal

[2] mesh with roughly half a million triangles

solution in the original graph. Though if the heuristics do hold up, there is a high likelihood to find at least a good quality solution.

However, the chance to find a good solution decreases as the difference between the original graph and the coarsened graph increases. This is problematic since the coarsened graph must be very small to find a solution quickly but the input graph might be very large. To mitigate the issue of the heuristic induced errors, the coarsening and re-projecting is done in several steps. First, progressively coarser version of the graph are generated, creating a chain of graphs with decreasing size. Once a sufficiently small version was created, a k-section is performed. This k-section is then projected back up the chain and refined at each step until the original graph has been reached again.

This approach has been demonstrated to provide good results for a wide variety of graphs. For meshlet generation, however, there are yet again no guarantees that the partition sizes are optimal. Naturally, unless the number of triangles in a mesh is exactly divisible by the target meshlet size, there will be triangles left over that will force a sub-optimal cluster size.

For meshlet rendering, the ideal solution in such a case would be to gather all left over triangles in a single sub-optimal meshlet. A less optimal solution would be to accept general discrepancies in meshlet size and distribute the remaining triangles over several of them. If this results in a meshlet being over the maximum size, it would need to be split into two smaller ones. Alternatively, the number of partitions could be adjusted such that it is a divisor without remainder for the number of triangles. This, however, is impractical if the number of triangles is not divisible by a number close to the target size. As hinted at by Karis, the graph partitioning algorithm in METIS requires special handling for cases as sometimes the target size is exceeded. Implementing a way to guarantee the ideal solution for uneven divisions, would be an aspect where the multi level partitioning scheme could be improved upon.

6.1.4. Direct Clustering Approaches

The quality of the heuristics used for grouping graph nodes during the coarsening phase of multi level graph partitioning has a major impact on the quality of the end result. Fortunately triangle meshes – compared to arbitrary graphs – provide additional information such as measures for (physical) distances, surface area, etc. which can be used to improve these heuristics.

Since a graph node represents a triangle in the triangle mesh, a grouping of graph-nodes is equivalent to constructing a meshlet. This means that at the lowest level of the coarsening-chain the merged nodes may already be a valid split of a mesh into meshlets. The most trivial implementation would be to start at one triangle and add connected triangles into the meshlet group until the group is at the target size. Then, a not-yet assigned triangle is taken to start a new group and the process is repeated until no more triangles are left without a

group assignment.

Integrating more information obtained from analysing the triangle mesh leads to a variety of ways to construct the meshlet groups. This can be done for example by building a kd-tree [20] and using spacial relations ships in this tree, apply partitioning via k-medoids [21], prioritizing based on shared vertices [22] , optimizing for the radius of the cluster bounding sphere [22] or using hybrid approaches of different techniques.

However, most of the current meshlet generation techniques do not focus on building a meshlet hierarchy. As a consequence, most do not achieve the desired meshlet characteristics. In essence, this stems from the lack of a step wise re-projection with intermediate refinements to the partitioning. On an abstract level, the current direct approaches for meshlet generation project a k-section from the most reduced version of the graph directly back to the original. Due to the large jump, any errors introduced by the heuristic are inflated. The results are, thus, not a good approximation for a minimum width k-section.

6.2. Algorithm Proposal for Meshlet Generation

Building meshlets by incrementally adding triangles and perhaps refining local boundaries does not optimize directly for the characteristics that are desired of meshlets. Addressing this issue, a novel direct approach to splitting a mesh into meshlet has been develop for this thesis. This approach is based on the following premise:

We[3] can take advantage of the fact that a 3D mesh is the approximation of a continuous surface to find a specialized solution for the graph partitioning problem.

6.2.1. Continuous Case

Assume there is a 2 dimensional convex shape. For simplicity this shape also contains no holes. This shape is imagined to have a form of *mass* that is equally distributed across its surface. For this shape, we want to find a cut such that the shape will be split in two connected halves with equal mass.

Consider the specific case of a rectangle with side lengths w and h where $w > h$. Finding a cut with minimum length that optimally bisects this shape can be found trivially by first finding the middle points of the lines with length w and connecting them with a straight line. This cut will then have length h.

It is well known that for an euclidean space the shortest path between two points is a straight line. Restricting the search space of possible bisection-cuts of the rectangle to

[3] "we" = people interested in finding a solution to this problem

straight lines (from one point on the border of the shape two another), an observation can be made: The lines all intersect at the center of mass (or centroid) of the rectangle.

The observation that a straight line bisection intersect the centroid stays true even if the rectangle is deformed into an arbitrary shape. This has to be the case since a straight line not passing through the center of mass would necessarily need to have more mass on one side than on the other. Thus, the line would not cut the shape into equal mass parts.

Crucially, using the opposite direction of this relationship, it is clear that every bisection-cut can in principal be found if the center of mass is known. This generalized also to shapes that have holes or are concave in which case the center of mass may lie outside the area of the shape.

6.2.2. Unbalanced ratios

A straight line cut that divides a shape into two parts of a specific ratio is not as easily found as an exact bisection. Fortunately, we can use a balanced bisection cut to find a cut for any other ratio.

Assume we have a balanced bisection cut going through points A(anchor) and P(partner). The anchor point will remain fixed as part of the ratio-cut. To find a cut of a specific ratio, a binary search strategy can be employed: Pick a point P' along border that is not exactly P or A and use this point to define a new cut. This devides the path along the border into three distinct sections.

If the ratio cut by $A - P'$ is too large, pick a point between P and P' as the new P'. Otherwise, P will become P' and a point between P' and A will be selected to be the new P'. Repeating those steps, the ratio will converge to the desired one.

Note that there are always two possible points where P' would result in the desired ratio. One on the path from A to P and another on the opposite direction from P to A. Depending on the shape, those two cuts may not be of the same length.

6.2.3. Application to 3D meshes

A 3D mesh representation is an approximation of a continuous surface. This implies that any characteristic of the surface should be an approximate characteristic in the mesh. In our use case, this means that if a surface represents a 2D shape and has a straight-line bisection going from one point on the shape's border to another, a bisection-cut in the mesh should be located at approximately the same location. Further, the minimum width bisection-cut in the represented shape has a high likelihood to correspond to the minimum-cut in the mesh representation.

However, finding the minimum width straight-line cut in an arbitrary shape is not easily done. While it has been established that any cut must go through the center of mass of this shape, there exists an infinite amount of possible cuts through the center of mass, all of which would need to be checked. Here is where the mesh being an approximation actually becomes an advantage. The infinite number of points on the shape are already quantized as the vertices of the mesh. So only the cut going through these vertices can / need to be checked. With this, the number of candidates to reduced to a finite amount.

The next issue that arises is how to find this set of straight line cuts through the center of mass in the mesh representation. The mesh would first need to be mapped onto a 2D plane and arranged such that every triangle represents an equal amount of mass, specifically, the triangles would need to be shaped to have the same surface area. The center of mass could then be found by averaging the mass centers of each triangle and the vertex closest to it chosen to be its representation.

Unfortunately, performing this transformation for such an embedding is not something that can be accomplished easily. It would also only work for surfaces that do not form a closed 3D shape since any embedding of such a surface would necessitate to fold the mesh onto itself in the plane projection. Instead, of attempting such a transformation we will make use of relationships that can be queried only by traversing the edges and vertices of the mesh.

6.2.4. Prerequisites

Reasoning about the mesh as if it was a 2D shape requires a few assumptions to hold true:

First, the 3D mesh must have a single hole and not be a closed shape such that it can be spanned into a 2D plane. The vertices on the border of this hole define the border of the 2D shape. For practical purposes in case multiple holes are present, connect them by finding the shortest path between them. In case the mesh is watertight, i.e does not contain a hole, somehow create such a hole by removing a connected set of triangles.

Second, an edge is shared by no more than two triangles. Edges on the border can be identified by connecting only to a single triangle. Edges with more than two triangles have impossible geometry which cannot be reasoned about. A pre-processing step may be applied to separate such triangles, if they occur.

Third, triangles are always connected by an edge or not at all. Triangles that share only a single vertex must be treated as being fully separated. This is easily ensured by analysing the mesh-graph for connected components and treating each component separately.

6.2.5. Analogies

The closest analog for a straight line in a mesh representation is the shortest path between two vertices – measured in the number of edges that are traversed. However, this only holds true to a certain extend. For instance, several distinct paths from one vertex to another can all have the same length though clearly only one would correspond to the actual straight line. Since actual distance information is not available or not useful without the aforementioned transformation of triangles to all have equal area, a shortest path in the mesh has a high chance to correspond to a curve instead.

When a curve in the shape is represented by a shortest path in the mesh, it is analogous to looking at a mountain on a map and searching for the shortest path from one end to another. On the flat map, the line between the points would go over the tip of the mountain, but in 3D reality, the mountain might be so tall that it is a shorter way to just walk around it. However, to bisect the mountain, the straight path through the tip of the mountain as marked on the map would need to be taken. If the mountain peak is on the actual straight line we want to walk, then the shortest path from our location to the mountain peak and then the shortest path to our destination would compose the correct path to take.

Continuing with the mountain analogy, our mesh is a mountain range to traverse where a denser collection of triangles at specific parts builds one such mountain. Those mountains can also have smaller mountains themselves, corresponding to local triangle density changes within a larger region. Of course, there can also be valleys representing regions with lower triangle density. Though even and uneven mountain range can be bisected by finding the center of mass.

6.2.6. Mountain Cleaver Algorithm

For finding a straight line path through the mountain range, a final insight is needed: On any path there is exactly one point exactly in the middle between start and end. One of these paths will be the straight line we are searching for.

With the path found we can now returning from the mountain analogy and formulate a new algorithm for splitting a triangle mesh into meshlets:

1. Order Border Vertices.

   The vertices forming edges with only a single triangle attached to them must be ordered to form a ring with a consistent direction. For this, one such vertex is picked as the starting point. Any vertex on the border will connect to exactly two other vertices that are also on the border. One of them is chosen to be next in sequence. This vertex will then connect to another border vertex and sequence numbers can be assigned until the first vertex is reached again.

The border can be traversed in increasing or decreasing direction by moving to increasing/decreasing the sequence numbers of the border vertices. If increasing the sequence number would go beyond the number of border vertices, it is wrapped around based on modular arithmetic.

2. Find Start-and End-Point Candidates

   We know that the cut we are looking for will go from one point on the border to another through the center of mass. Since the center of mass is unknown, it is also unknown which points form such a line. Testing every possible combination of vertices along the border would result in a large number pairs, scaling quadratically with the number of border vertices. Most of these pairing would belong to a cut of different ratio than what is desired, so they should be filtered if there is no reason to believe that the pairing would correspond to such a line.

   In a similar fashion to narrowing down the arbitrary cut-ratio in the exact shape, we can find for any vertex on the border a partner for the lower bound and the upper bound. Specifically, a partner vertex where the shortest path is a cut as close to but still below the desired ratio and a partner vertex where the shortest path is a cut closest to but still above the ratio. It is clear that those two points will be directly adjacent on the border. If this were not the case there would be trivially another shortest path between the upper and lower bound which could be used for a tighter bound in either direction.

   For the pairing process to work, we need to know in which "direction" to search for a tighter upper or lower bound. This is where the order of the border vertices comes into play: We can define the lower bound side as the side where triangles are connected to edges who are between the start-and end vertex. The direction is defined by order of increasing sequence numbers (and applying wrapping if necessary). All triangles reachable by those triangles without crossing an edge that is part of the cut we are attempting to measure are also considered to be on the same side. This side will be referred to as the *covered-side*.

   We are interested in the "flipping point", i.e. the point where the covered-side goes from being covering less than desired to covering more. To find this point, we can either linearly go along the path until the flip is encountered, or repeatedly pick the sequence number in the middle of the current above and below cuts and adjust the bounds accordingly.

   To ensure a valid result, any shortest path from the anchor vertex to the tested partner is sufficient as long as the path does not contain any other border vertices. Such a path must not be considered to evaluate a pairing since it would better correspond to a different pair or separate the mesh into more than only two connected regions.

3. Find middle points to straighten the cut.

   Choosing any shortest path from start to end is likely going to represent a curved path around a mountain instead of the straight line we desire. Making use of the fact that any path has a point exactly in the middle between start and end, we find vertices in the mesh that have the same shortest distance to both the start and the end vertex These vertices are candidates for the middle of the straight line.

   Note that the exact middle point might actually lie on an edge or inside a triangle. To account for this, the candidate for a middle point can also be an edge. Such a middle edge can be found by comparing the distances of two vertices. If the distance to start of one vertex is the same as the distance to end of a connected vertex and vice-versa, the edge between them is such a middle edge.

4. Find bounds on maximum and minimum coverage

   Due to mountains potentially housing smaller mountains within, the shortest paths to and from the middle point may also walk around a triangle mountain instead of representing a straight line path.

   Again, we try to find a "flipping point". This time, we want to find out how much area a path going through a middle point can take. Essentially measuring the area under the curve spanned by this path. There will once again be a point where the covered area goes from covering too little to covering too much. These are the middle points that are closest to being on a straight line path.

   Once the selection has been narrowed to two middle point candidates, the candidate, the minimum coverage area is searched for the candidate that exceeds the desired coverage. In case the minimum coverage is below the target coverage, this path is preferred over the maximum of the candidate with below-target coverage.

   During this search, the desired amount of coverage might accidentally be found in which case the search can terminate already.

5. Choose sub-region to narrow down

   There will now be up to two sub-regions, one spanned by the start and middle vertex ($s - m$-region) and one spanned by the middle and end vertex ($m - e$-region). The area of active search contained within the previously found paths found in the previous step. Each region may contain their own set of mountains / triangle dense regions, so the paths of minimized and maximized area may meet and diverge several times, forming a sort of double-helix pattern.

   The covered area below the target coverage will now be referred to as the *base*-region.

This region will be expanded until it has reached the desired partition size. For this, triangles the $s - m$ or $m - e$ region will be added to this base.

There are now four constellations that may occur which are relevant to decide which triangles to add to the base region:

- (*base* + $s - m$ < target size) and (*base* + $m - e$ > target size)

  Adding the $s - m$-region would not push the coverage size over the target size and there will not be a shorter path that also covers as much area in this region. Hence, we choose to be greedy and integrate the entire area of the $s - m$ region into the base region and refined the search in the $m - e$-region.

- (*base* + $m - e$ < target size) and (*base* + $s - m$ > target size)

  Same as the previous case, but with the roles reversed.

- (*base* + $s - m$ > target size) and (*base* + $m - e$ > target size)

  Adding either regions would push the base-region size above the target size. In this case, we discard the option to add any triangles from the larger of the two regions and refine the search in the smaller region. The smaller regions is chosen because fewer triangles generally tend to have a shorter path separating them.

- (*base* + $s - m$ < target size) and (*base* + $m - e$ < target size)

  Adding either region to the base-region would not go beyond the target size. In this case, the larger of the two regions is absorbed into the base region and the search will be refined in the smaller regions.

Note:

Choosing between either of the two sub-regions avoids the complexity of needing to deal with the exponential amount of combinations that would arise if one would try to refine the search in both sub-regions simultaneously. This means there is a chance that this approach may not find the actual shortest path for a bisection.

6. Refine the Sub Region

   Depending on which region what was chosen for refinement, the previous middle point will now serve as either the new start (m-e-region) or the new end (s-m-region). The same process of finding middle points to find the straightest line-cut can now be recursively applied to the chosen sub-region, incrementally adding to the base region until the target size is exactly reached.

   Edges that connect triangles in the covered region from those in the uncovered region of the mesh are then the edges of the cut.

7. Choose best cut

   Every pairing of start and end point candidates results in a unique cut. Of these cuts, the cut with the fewest edges can be chosen.

8. Recursively split partitions

   The above process splits the mesh into two parts. On each path the same process can now be applied to further split the mesh until it has been divided into the desired amount of meshlets.

6.2.7. Results and Comparison

The process is illustrated in the appendix (see Figure 6 and 7). Figure 3 shows the results of the mountain-cleaver algorithm to the popular Stanford Bunny model [23]. The model has 69451 triangles that are grouped into meshlets of 64 triangles each. Only a single meshlet has eleven triangles which is the optimal amount since the number of triangles is not evenly dividable by 64.

It can also be observed that the boundaries between meshlets are mostly straight with the exception of a few cases where a single triangle is needed to ensure even group size. There are, however, parts of some meshlets that are only a single triangle wide. This is not optimal since it creates a long shared boundary between three meshlets at the same time. The suspected reason for this is that the cut is locally optimal and indistinguishable from a similarly optimal cut that would have a better global result. An additional refinement step could be added to address this issue.

In Figure 4, the meshlet generation using the mountain-cleaver algorithm are put side by side to other approaches (images of the other meshlet generation approaches where taken from Jensen et al. [22]).

There are two observation that can be confirmed visually. First, other approaches have a tendency to creates many *spikes*. Spikes are triangles that share more than one edge with another group. Such a triangle has all three of its vertices locked during the simplification step, so it has a high chance to remain "as is". Fewer spikes, as in meshlets generated by the mountain-cleaver algorithm, are therefore preferable for building a meshlet LOD-hierarchy.

Second, other approaches tend to create small, trapped regions. This stems mostly from the process of expanding meshlets until they have reached their target size. Creating meshlets this way has a high chance to leave out some triangles in between three or more fully formed meshlets. This then leads to meshlets with lower than the desired size needing to fill the space. This is by construction avoided in the mountain-cleaver algorithm resulting in significantly fewer meshlets with sub-optimal size.

Figure 3.: Stanford Bunny [23] divided into meshlets (64 triangles) using Mountain-Cleaver-Algorithm

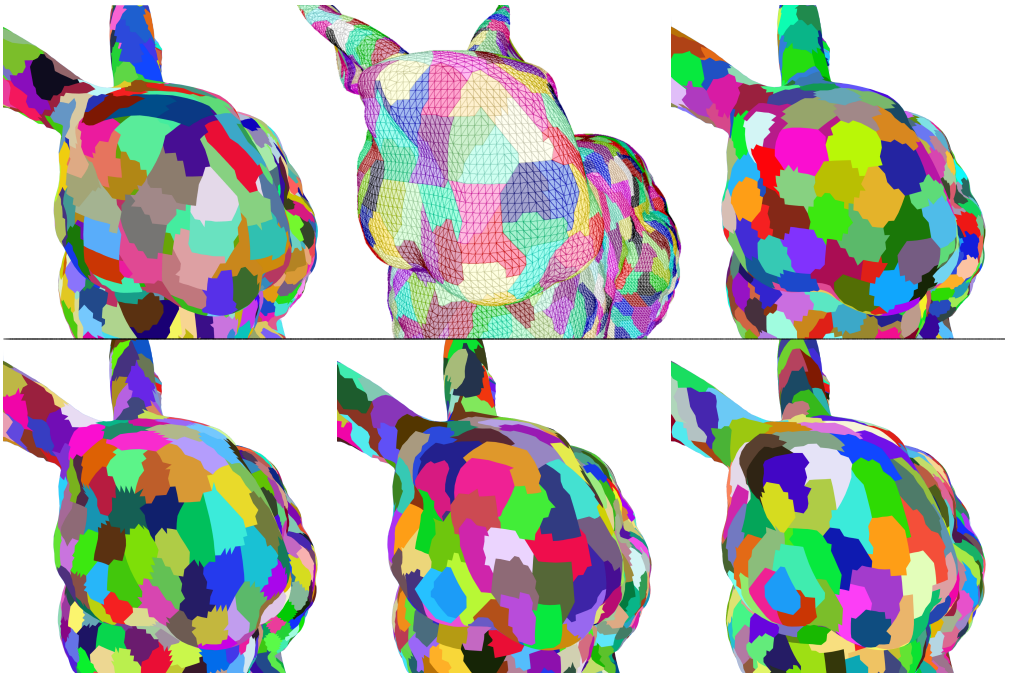


Figure 4.: Visual Comparsion of open Meshlet generation approaches. Top Left: Nvidia (Kubisch) [7], Top Middle: Mountain-Cleaver (own), Top Right: Zeux Meshoptimizer(Kapoulkine) [20]. Bottom Left: k-medoids, Bottom Middle: Greedy (Jensen, et al.) [22], Bottom Right: Bounding Sphere (Jensen, et al.) [22]; Images of other approaches taken and modified from (Jensen et al.) [22]

7. Merge and Simplify

7.1. Meshlet Grouping

7.1.1. Quality Metric

Once the first set of meshlets has been created, the meshlet LOD-hierarchy can be built. This is done by repeatedly grouping meshlets, applying a simplification operation on all triangles within the group and then splitting the simplified triangle cluster into separate meshlets again. How groups of meshlets are formed will influence how the hierarchy of meshlets will be structured and how flexible the hierarchy will be. So it is important to find a good strategy for building these groups.

Grouping meshlets is once again a graph partitioning problem. The graph nodes are now the meshlets and the graph-links now represent the shared border between meshlets. Although this time, the parameters that should be optimized have slightly changed:

- **Meshlets within a group must be able to simplify together**

  For grouping meshlets, connectivity is not optional but strictly required. Otherwise the simplification operation will not be able to clean up any shared boundaries, voiding any benefits of this LOD system. This also implies that meshlets need to be connected by at least two shared edges for a simplification operation to be applicable to both.

- **Groups should simplify to fully filled meshlets**

  This is to ensure that meshlets of the new LOD will be rendered as efficiently as possible and not leave any inactive lanes in the GPU's SIMD units when processing the triangles. Increasing memory storage and bandwidth demands by introducing padding triangles will also be avoided if the meshlets are well formed.

- **Group members should have similar LOD-cut-values**

  The LOD-cut-value of the meshlets generated by the merge-and-simplify operation is always monotonically increasing between levels of the hierarchy. So the LOD-cut value of the simplified meshlets will be larger than the highest LOD-cut-value of the source group members. Ensuring a similar LOD-cut-values within a group helps to maintain an even amount of detail and avoids to early elimination of patches high density details.

- **Groups should be as small as possible**

  Fewer members per group give a larger flexibility for finding a cut in the LOD hierarchy. This in turn provides finer granularity when it comes to selecting meshlets for rendering. The larger a group becomes the more meshlets will have to be rendered for the same amount of local detail. This also increases the risk that some areas will have to render with unnecessary high amount of detail which increases processing cost to ensure a high image quality or leave some areas under-detailed maintaining low processing effort at the cost of image quality.

As a consequence of these optimization criteria, the optimal number of partitions can no longer be predetermined. This is because there is no way to tell beforehand how large individual groups need to be for them to be able to simplify properly.

Nevertheless, finding a minimum width k-section of the meshlet-graph still strongly correlates with the desired characteristics of the grouping result. If meshlet groups have minimal shared borders between each other, the meshlets within a group will be strongly connected and, thus, have many options for simplification. Strongly connected meshlets also have a high likelihood of having a similar level of detail. So all aspects of an optimal grouping are benefited by finding a minimum width cut. As a result, a general solver for graph partitioning, e.g. the multi-level partitioning scheme found in the METIS library, can be used to find a good solution for meshlet grouping. However, since the best number of groups is unknown, potentially limiting the partitioning algorithm, approaches who directly attempt to optimize for the desired characteristics are worth exploring.

7.1.2. Wave Function Collapse

Exploring other areas of scientific study can lead to new ideas for one's own field. Exploring the field of quantum mechanics, Gumin [24] found inspiration in the behaviour of quantum particles:

A collection of particles starts out with every particle being in a super position of the all possible states this particle could take on. Observing or measuring a particle results in the collapse of its wave function and from all possible states in the super position, one specific one is chosen based on a certain probability described by the wave function. Particles are entangled with one another so the collapse of one super position changes the wave function of other particles, potentially leading to a cascade of collapses.

Based on those ideas, Gumin presented the Wave Function Collapse algorithm which was originally used to create a procedural tile map. A tile map is a grid of cells where each cell can be one of a limited set of states (called tiles). Each tile can connect only to specific other tiles when building the tile map.

In the wave function collapse algorithm, each cell in the tile map starts out in a super position of every possible tile. When a specific tile is chosen for a location, the set of possible tiles in neighboring locations are updated to include only tiles that may connect to it. The map is then built by repeatedly collapsing the cell with the lowest entropy, meaning the lowest amount of possible tiles to choose from, and deciding on one specific tile.

The ideas of the wave function collapse algorithm are not limited to 2D tile maps, but can generalized to apply to arbitrary systems with dependant constraints. For instance, we can formulate the meshlet grouping problem as such a system and apply the wave function collapse algorithm as follows:

- The wave function is described by which meshlets might be assigned to a group. Initially, there are as many groups as there are meshlets.
- A meshlet has a certain probability to belong to a different group. This is based on the strength of the connection to other groups and on whether the connected group is "satisfied". The strength is further influenced by the number of shared edges between the meshlet and meshlets already known to be part of the group.
- A satisfied group is a group that has gathered enough meshlets to allow a simplification to a whole number of fully filled and simplified meshlets. Initially, meshlets not yet assigned to a satisfied group cannot join such a group.
- The entropy of a meshlet is measured by the number of connections to other open groups. The meshlet with the lowest number of possible groups to chose from has the highest priority for being collapsed. When two meshlets have the same number of connections, the meshlet with the strongest connection is prioritized. For remaining cases, the order can be determined freely.
- When a meshlet's super position is collapsed, the entropy of surrounding meshlets is reduced by updating which groups are connected and how strong the new connections are. If a meshlet joins a group and this results in creating a satisfied group, the connections other meshlets had to this group are severed.
- If after the first round of collapsing some groups are still left unsatisfied, another round is started where it is once again possible for the unsatisfied groups to join any another group. This is repeated until every group is satisfied.

If the first results are unsatisfactory, randomization can be used to find other possible outcomes and chose from the best overall. However, this is likely unnecessary since building the hierarchy this way is to a certain extend self-correcting. If a long border was not simplified on one level, there is an increased chance it will be used for group merging in the next level.

The self correction factor of the hierarchy building can also apply to other approaches, so it seems safe to assume that any reasonably good solution to the problem will yield an adequate result. Unfortunately, due to the lack of sample size both in terms of models to test, other established approaches as well as lack of resources for conducting a more in depth analysis, no concrete data about measurable quality differences can be provided.

7.2. Meshlet Simplification

7.2.1. Simplification Operations

With a meshlet group formed, the triangles span a small triangle mesh that now needs to be simplified. There are three general kinds of simplification operations that can be applied [25]:

1. Vertex Removal

   A vertex is chosen to be removed alongside all triangles connecting to this vertex. The resulting hole in the mesh is then re-triangulated.

2. Edge Collapse

   Two vertices connected by an edge are contracted into a single vertex. All triangles attached to one of the two vertices are attached to the collapsed vertex.

3. Triangle Removal

   Three vertices are removed by collapsing them into a single one. Triangles connected to either of the three vertices are attached to the new vertex.

7.2.2. Quadric Error Edge Collapse

Triangle mesh simplification is a well researched topic with many established solution for high quality mesh simplifications. One such widely employed solution is edge collapse based on a quadric error metric, originally introduced by Garland et al.[26].

With edge collapse simplification, the vertex resulting from the collapse should be placed in a way such that the error metric is minimized. For measuring the error, each vertex is associated with a *quadric*, which is a 4x4 matrix that encodes the squared distance to the planes of the triangles meeting at the vertex. Quadrics of connected vertices can be combined with simple addition to obtain the quadric for the edge. The structure of a quadric allows to analytically find the placement for a vertex such that it minimizes the error as calculated by this quadric. Improvements to the error formula have been made by also factoring other attributes [27] such as the area of the triangles, normal vectors, etc.

Quadric error edge collapse has been refined to achieve mesh simplification with a minimized difference in perceptual loss of quality. However, comparatively not much research seems to have been done to generate models such that they are more efficient to render on modern graphics cards. [1]

7.2.3. Max Area Vertex Removal

The hierarchical meshlet LOD system allows to pick meshlets such that the error on screen is almost imperceptible. However, triangles need to relatively small to accomplish this. As already established (see chapter 4), for triangles that only partially cover a 2x2 pixel quad, fragment shading work is wasted in form of helper invocations.

Helper invocations occur at the edges of triangles. So for a group of triangles to minimize the number of helper invocations means to minimize the length of the shared borders. Assuming a shape has a distinct border, the edges not shared by two triangle in the group form a polygon in 3D space. Minimizing the length of edges needed to triangulate this shape is known as minimum weight triangulation. This is another NP-hard problem [28], although good results can be achieved using heuristics.

Unfortunately, edge-collapse approaches for simplification tend to result in poor triangulation when measured in terms of total edge length. Edges of two vertices will all be connected to the same vertex after the collapse, creating "star"-like patterns. In an experiment performed by Persson [29] to evaluate the performance of different triangulation schemes, star and strip like patterns were shown to require significantly more processing time than what he called "max area" triangulation. In the max area approach, the triangle with the largest are was greedily taken out of the polygon until it was fully triangulated.

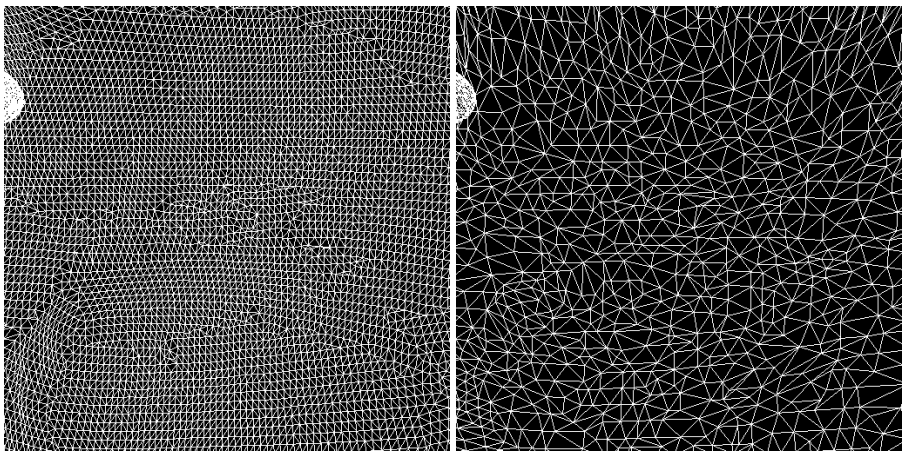
Given that Persson's max area approach was demonstrated to be significantly better for processing on the GPU than other triangulation schemes, the re-triangulation step of a vertex removal operation should aim to achieve a similar max-area result. Additionally, there should be a way to measure the error of the vertex removal such that vertices can be prioritized in the removal process to minimize the global error.

To address the nonexistence of such an algorithm, a vertex decimation approach can be adapted:

- Error is measured as the sum of volumes of the pyramids which are formed with the triangles created by the triangulation as the base and the removed vertex as the tip.

  The improved quadric error metric incorporates both the distance from the planes of a triangles as well as the area of the triangles. The area of a plane multiplied by the

[1] Understandably, there is little incentive to do so since visual quality usually tends to outweigh some minor potential performance improvements



(a) Wire frame – base mesh (b) Wire frame – simplified

Figure 5.: Max area vertex decimation applied to Stanford bunny mesh

distance of a point from the plane is a measure for a volume. Instead of measuring the volume change introduced by moving a vertex, we instead check the volume change of a re-triangulation.

- Triangles for the re-triangulation are formed by vertex with the longest distance from the to be removed vertex and two vertices connected to the chosen vertex.

  The number of vertices attached to a single vertex tends to be relatively low, so a brute force search for the best triangulation would be viable. However, This triangulation has a high risk of spanning triangles over areas that where previously not filled. Always choosing the furthest vertex to be the base of a new triangle avoids this issue by choosing triangles where only in an area that was already spanned. This choice of vertex also has a high chance correlating with short edge between the newly connected vertices. If additionally, vertices are prioritized when they were not previously connected to a vertex involved in a triangle formation, similar results to the max area triangulation are achieved (See figure 5).

8. Candidate Selection

For any given scene, there are typically many more meshlets present than those who are visible and even fewer than those who should be selected to be drawn. The information which meshlets should be drawn is primarily determined by which instances are at least partially visible and which level of detail the instance should have.

In the case of discreet LODs, an instance that passes the visibility check would emit one draw command for the mesh representing the most fitting level of detail. This one to one mapping of input and output makes it simple to implement on the parallel architecture of a GPU. When using a hierarchical meshlet LOD system, however, it is not a 1:1 mapping anymore. Now, one instances may map to arbitrarily many meshlets for which a cut in the LOD hierarchy must be found.

As discussed in Chapter 3, emitting draw commands for any amount of arbitrarily sized meshlets is possible in current graphics processing units. However, GPUs currently do not expose any way to emit arbitrary compute work in a similar fashion. Specifically, there is no equivalent to multi-draw-indirect-count for compute which would provide a way for a compute dispatch to receive a work ID handle from the scheduler.

With the lack of an equivalent to multi-draw's draw ID, there are only three general options left available:

- Accept empty dispatches.

  Emit on the CPU side a command for the worst case scenario. For example one command per possibly visible instance. The GPU would then write a zero value for instances that should not be rendered. The obvious downside being the overhead introduced by processing dispatch commands that do not result in active work being done.

- Communicate the counter variable to the CPU.

  This would allow avoiding empty dispatches but would require synchronization between CPU and GPU. Circumstantially, this approach might more or less viable (see 8.4).

- Emit a single, large dispatch.

This approach is similar to dynamically generating the index buffer and emitting a single indirect draw command. Some way to compactly emit compute work in a fixed format would be required for this approach.

Due to the drawbacks of the first two approaches, the most viable option is to emit a single (indirect) compute dispatch. The question then is how to best distribute the work to the available compute units.

8.1. Fill whole queue

If the number of instances and meshlets is not very large, a brute force solution could be employed. The parallel nature of LOD-cut selection makes it possible to test every meshlet individually. However, this is unlikely to result in good performance because meshlets are shared by different instances but each instance might prefer a different LOD-cut. So an entire meshlet hierarchy would need to be retested for every instance that refers to it. Commands would also need to be preemptively emitted for meshlets with no visible instances, making the naive approach prohibitively inefficient.

To ensure that only meshlets for visible instances are tested, an instance which is selected for rendering could push a combination of meshlet ID + instance ID to a queue for the next dispatch to perform the selection. This would be done for every meshlet of the instance's mesh, creating a 1:1 mapping between meshlet candidates and dispatched compute threads.

The issue with this approach is the large amount of data that needs to be written per thread. Further, there is a high risk of imbalance in the work done per instance thread. Due to the one to many relationship between instances and their meshlets and the varying size of meshlet hierarchies some threads may be occupied with emitting large amounts of meshlet works while leaving other threads running idle. The GPU would then stall on a few threads and not make good use of the available parallelism. Instead, pushing work onto a queue should ideally be done in parallel by many threads.

8.2. Nanite – Persistent Threads

The solution in the Nanite implementation makes use of a bounding volume hierarchy (BVH) to accelerate meshlet selection. The BVH is structured based on the LOD-selection-value of a meshlets's parent instead of on the LOD-hierarchy.

The selection process (presumably) starts with each visible instance pushing a BVH root node for its mesh on a queue. Processing a BVH node will result in either the nodes child nodes being pushed to the work queue or a number of meshlet being selected for rendering.

Nanite's first proposed implementation processes the BVH nodes level by level. So first the root nodes are processed which push work for the next level onto a queue. Another dispatch then processes all the items in this queue until the last level of the BVH has been reached. Due to the lack of emitting a variable amount of dispatch calls on the GPU, this implementation executes the worst case scenario number of dispatch calls. This results in some dispatches being empty and thus some wasted work in processing the commands.

Nanite's alternative implementation on the other hand does only a single, large dispatch to process all of the work. By dispatching as many threads as required to saturate the GPU and manually managing a job queue onto which threads will dynamically push and put work items, the ability to schedule work from compute units is effectively emulated in software. Karis calls this the "persistent thread model".

The persistent thread model relies on scheduling behaviour of the GPUS's work scheduler. As Karis noted, this behaviour is not specified by any standard, so it relies on experimentation to find out on which GPUs this approach is viable. Karis further notes that so far all GPUs relevant to them exhibit the required behaviour. However, for Nanite, this approach was likely only validated for desktop and console GPUs. It is an open question whether mobile GPUs also generally have the required scheduling behaviour.

8.3. Simple GPU task system

Nanite's persistent thread model uses a multi-producer-multi-consumer queue that relies on locks for synchronizing which necessitate the specific GPU scheduling behaviour. Though what if the need for dynamic producers could somehow be eliminated?

If meshlets are arranged such that their LOD-cut values are within a certain range of each other or such that meshlets on the same LOD level are next to each other in the global buffer, then visible instances could make a decision about a range of meshlets at once. When in traditional discreet LODs an instance would choose a specific LOD level, the instance would now consider a range of LOD levels and emit a span of meshlet to check for each. The range of considered LOD levels should be as tight as possible, ideally only being the levels directly above and below the LOD-cut. This is likely to be achievable when meshlets on the same level have similar LOD-cut-values.

Once the data has been arranged this way, a visible instance can now emit a compressed work item containing only the following:

- Instance ID to access instance transform
- Starting meshlet ID
- Number of meshlets in queue

- Completion counter (initially set to zero)

The counter for the number of threads would still be incremented by the number of meshlets per queue. Though this will now be done in large chunks instead of one by one.

To then work through the job queue, the following steps for a meshlet selection thread are proposed:

1. Find a work item.

   The queue of work is much smaller than the number of threads so the thread invocation id needs to be mapped to fit in the range of the queue. If neighboring thread IDs should prefer to work on the same span of meshlets, divide the invocation id by the queue size and round the result to the nearest integer. Alternatively, threads with neighboring IDs can be assigned to different meshlet ranges by using the modulo operation instead.

2. Atomically increment the completion counter of the work item.

   This gives the index of the meshlet that will be processed by this thread.

3. Maybe try again.

   If the completion counter is higher than the number of meshlets in the range of the current work item, proceed to find another work item. and try step 2 again.

4. Process meshlet

   When a counter has been incremented without going over the range, fetch the data for the meshlet test and emit draw command if the meshlet should be drawn.

8.4. Hardware specific: Shared Memory

All previous solution assumed that the goal is to have a completely GPU driven pipeline. While this is often desirable in cases where the processing capabilities of the GPU vastly exceed those of the CPU and/or there is a major communication overhead that needs to be avoided, this might not be the case for some systems with low overhead such as those with a unified memory architecture.

The CPU always has the ability to emit commands for the GPU. The issue is that this requires synchronization between the two processing units. Between instance selection and meshlet selection the GPU needs to wait for the CPU to emit the commands and the CPU previously had to wait for the GPU to finish instance selection. This could be an acceptable trade-off depending on the specific application, for others, some mitigation would need to be found.

To avoid GPU idle time, some other work has to be already queued up. One solution to this could be to interleave work for the current frame with work for the next frame. In this scenario, the GPU would do the instance selection for the next frame while still waiting for the commands needed to render the current frame. If neither GPU nor CPU fall behind their work schedule, this should allow for efficient rendering with the only penalty being a one frame increase in latency for the user.

8.5. Fragment shader for compute

GPUs technically already provide a way to dispatch a varying amount of compute work in the form the classic vertex and fragment shader pipeline [30]. The vertex shader places vertices which are used by the rasterizer to determine which and how many worker threads should execute the fragment shader.

Having the ability to do general computations in both the vertex and fragment shader as well as arbitrary buffer read-and write access provide in principle sufficient mechanisms to perform variable work thread emission. It would be theoretically possible to have the vertex shader perform the instance selection and the fragment shader the meshlet selection.

8.5.1. Emitting fragment shader invocations

As already discussed in chapter 4 on how triangle size affects performance, fragment shading work is emitted as blocks of 2x2 pixels by the rasterizer. Within these blocks, some threads might be inactive helper invocations who do not cover any actual geometry and thus do not contribute any results. Any approach utilizing fragment shaders for compute work should thus try to avoid them as best as possible.

For avoiding helper invocations, it needs to be ensured that only full blocks of 2x2 pixels are covered. Unfortunately, this greatly complicates the mapping between the input for a single fragment shader invocation and the corresponding compute work, because work identifiers now need to first be encoded to map to a 2 dimensional space which might need to interpolate across the area of a triangle.

Covering exactly a range of 2x2 blocks is a difficult task to achieve with the standard standard primitives, especially in combination with needing to emit a precise amount of work. Some hardware may expose support for a rectangle primitive, which would simplify this task substantially. However, even if support is advertised, it might be implemented by emitting triangles in the driver instead, so no actual benefit will be had.

Line primitives are also possible candidates for specifying ranges of work. If a variable line width is supported, this width can be adjusted to cover entire pixel grids. If changing the

width is not supported, it will require tests on specific hardware to determine how drawing lines affects the generation of helper invocations.

8.5.2. Emitting Vertices

The best shape of the geometry for fragment shader invocations influences how vertices should be generated and processed. How this in turn is best achieved depends on the capabilities of the target hardware.

For instance, if compute shaders are supported, vertex information can be appended to a buffer from each thread that has a visible instance. With an indirect draw command, the meshlet ranges could then be processed all at once. However, this requires a barrier between compute and vertex shader, so meshlet processing cannot start until instance processing is finished.

Instead, if geometry shaders are supported, the vertex shader would take on the role of the instance selection compute shader and output point primitives to the geometry shader. The geometry shader would then expand the point into a line segment and send it to the rasterizer for emitting the meshlet selection work. In this approach, processing of instances and meshlets could potentially be better pipelined. However, the geometry shader may need to comply with restrictions on processing order which could still introduce a stall in the pipeline. Also, memory might need to be allocated for the worst case scenario, specifically if all instances are visible and emit two vertices. Though it should be noted that this is also necessary for other approaches. If a limited form of dynamic allocation is possible or memory is directly passed on to the next stage, the geometry shader approach could even be more efficient in that regard. Obviously, this is hardware and driver implementation specific and should be investigated for any given use case.

9. Conclusion and Future Work

Building an efficient hierarchical meshlet renderer is not an easy task. However, with the means of analysis and with the algorithms presented in this thesis, an informed decisions can be made to ensure optimal usage of available graphics processing hardware.

A good solution to an otherwise NP-complete problem, the novel mountain-cleaver for mesh cluster generation, has been established. This algorithm, alongside the insight and tools of analysis provided for evaluating the quality of meshlet generation, should aid future research in developing even better meshlet rendering system.

For predicting the efficiency of hardware-accelerated triangle rasterization based on triangle size, a formula was presented. This formula was shown to provide good predictions of processing costs, making it a suitable basis for tuning for efficiency in meshlet processing.

It was also shown how existing techniques and algorithms, such as vertex decimation and the wave function collapse algorithm, can be adapted for the construction process of meshlet LOD hierarchies that are focused on rasterization efficiency.

In summary, the tools, techniques and insight gathered in this thesis could provide the basis upon which specialized approaches to hierarchical meshlet rendering are built.

For future work, it is planned to utilize the accumulated set of algorithms and analytic metrics to investigate the viability of constructing a optimized hierarchical meshlet renderer for a low powered, but feature rich mobile architecture, for example as found in the Raspberry PI 4 or 5. Research into this area could potentially expand the horizon of what real time rendering applications on mobile phones are capable of.

A. Appendix

Research code will be made available at `https://github.com/Estard/hierarchical-meshlets`.

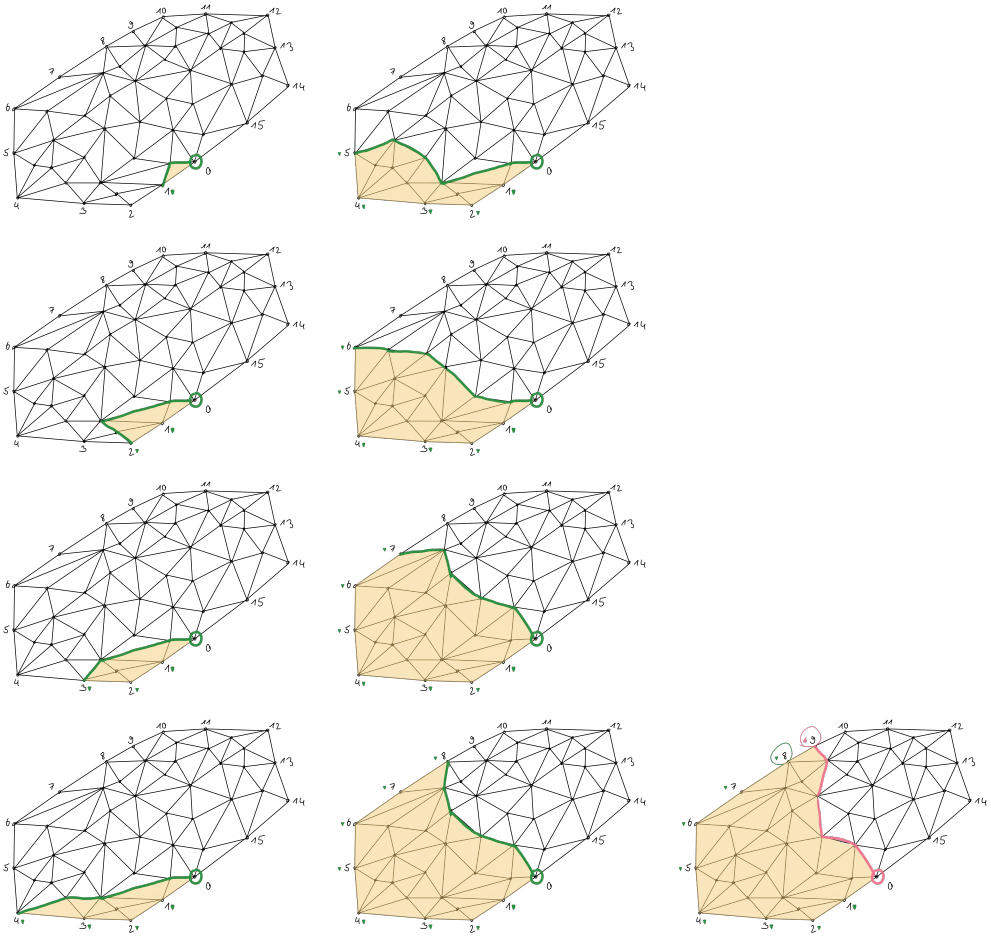


Figure 6.: Mountain Cleaver Algorithm: Partner search

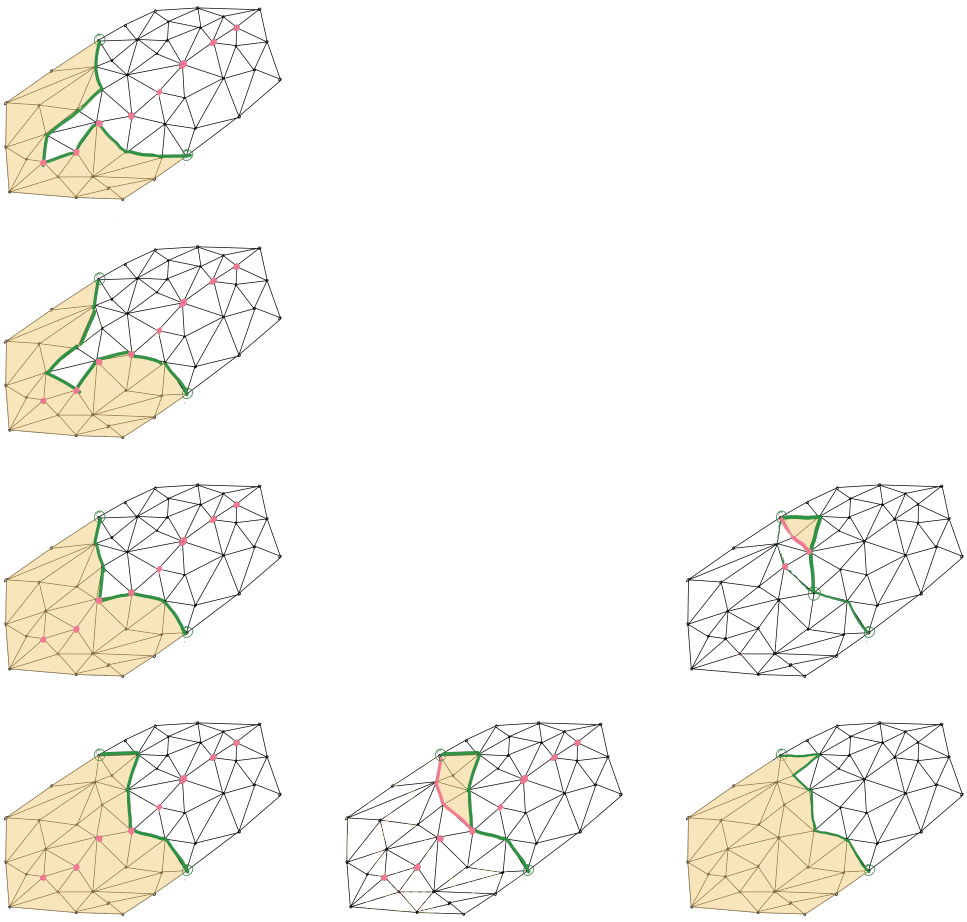


Figure 7.: Mountain Cleaver Algorithm: Sub-region narrowing

List of Figures

List of Tables

Glossary

BVH Bounding Volume Hierarchy. 52

CPU Central Processing Unit. 8

GPU Graphics Processing Unit. 1

Bibliography

[1] G. W. Brian Karis Rune Stubbe, "Nanite a deep dive," in *Siggraph*, 2021.

[2] NVIDIA Corporation. "Nvidia geforce gtx 980." (2014), [Online]. Available: https://www.microway.com/download/whitepaper/NVIDIA_Maxwell_GM204_Architecture_Whitepaper.pdf (visited on 05/14/2024).

[3] NVIDIA Corporation. "Nvidia ada gpu architecture." (2023), [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-V2.02.pdf (visited on 05/14/2024).

[4] Advanced Micro Devices, Inc. "Introducing rdna architecture." (Jun. 2012), [Online]. Available: https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf (visited on 05/14/2024).

[5] Advanced Micro Devices, Inc. "Introducing rdna architecture." (2019), [Online]. Available: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf (visited on 05/14/2024).

[6] S. A. Ulrich Haar, "Gpu-driven rendering pipelines," in *Siggraph*, 2015.

[7] C. Kubisch. "Introduction to turing mesh shaders." (Sep. 2018), [Online]. Available: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/ (visited on 05/14/2024).

[8] The Khronos Vulkan Working Group. "Vulkan® 1.3.285 - a specification (with all registered extensions)." (May 2024), [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html (visited on 05/14/2024).

[9] Microsoft Corporation. "Directxtex texture processing library." (Jan. 2021), [Online]. Available: https://github.com/microsoft/DirectXTex (visited on 03/12/2021).

[10] Apple Inc. "Metal shading language specification version 2.3." (Nov. 2020), [Online]. Available: https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf (visited on 03/12/2021).

[11] NVIDIA Corporation. "Nvidia dlss 3 maximale fps. maximale qualität. angetrieben durch ai." (2024), [Online]. Available: https://www.nvidia.com/de-de/geforce/technologies/dlss/ (visited on 05/14/2024).

[12] Advanced Micro Devices, Inc. “Amd fidelityfx™ super resolution.” (Jun. 2024), [Online]. Available: https://www.amd.com/de/products/graphics/technologies/fidelityfx/super-resolution.html (visited on 05/14/2024).

[13] I. Corporation. “Xess super sampling.” (2024), [Online]. Available: https://www.intel.de/content/www/de/de/products/docs/discrete-gpus/arc/technology/xess.html (visited on 05/14/2024).

[14] J. Thaler and T. Wien, “Deferred rendering,” *TU Wein: Vienna, Austria*, 2011.

[15] Kirill. “Why you should never use deferred shading.” (Sep. 2023), [Online]. Available: https://docs.google.com/presentation/d/1kaeg2qMi3_8nQqoR3Y2Ax9fJKUYLigPLPfdjfuEGowY (visited on 05/14/2023).

[16] C. A. Burns and W. A. Hunt, “The visibility buffer: A cache-friendly approach to deferred shading,” *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, pp. 55–69, 2013.

[17] O. Goldschmidt and D. S. Hochbaum, “A polynomial algorithm for the k-cut problem for fixed k,” *Mathematics of operations research*, vol. 19, no. 1, pp. 24–37, 1994.

[18] T. J. Schmidt, “On the minimum bisection problem in tree-like and planar graphs,” Ph.D. dissertation, Technische Universität München, 2017.

[19] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[20] A. Kapoulkine. “Mesh optimization library that makes meshes smaller and faster to render.” (2017), [Online]. Available: https://github.com/zeux/meshoptimizer (visited on 05/14/2024).

[21] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009.

[22] M. B. Jensen, J. R. Frisvad, and J. A. Bærentzen, “Performance comparison of meshlet generation strategies,” *Journal of Computer Graphics Techniques (JCGT)*, vol. 12, no. 2, pp. 1–27, Dec. 2023, ISSN: 2331-7418.

[23] W. Research. “"stanford bunny" from the wolfram data repository.” (2022), [Online]. Available: https://doi.org/10.24097/wolfram.91305.data.

[24] M. Gumin. “Wave function collapse algorithm.” (Sep. 2016), [Online]. Available: https://github.com/mxgmn/WaveFunctionCollapse (visited on 05/14/2024).

[25] J. O. Talton, “A short survey of mesh simplification algorithms,” *University of Illinois at Urbana-Champaign*, 2004.

[26] M. Garland and P. S. Heckbert, “Surface simplification using quadric error metrics,” in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, 1997, pp. 209–216.

[27] H. Hoppe, “New quadric metric for simplifying meshes with appearance attributes,” in *Proceedings Visualization’99 (Cat. No. 99CB37067)*, IEEE, 1999, pp. 59–510.

[28] W. Mulzer and G. Rote, “Minimum-weight triangulation is np-hard,” *Journal of the ACM (JACM)*, vol. 55, no. 2, pp. 1–29, 2008.

[29] E. Persson. “Triangulation.” (Jan. 2009), [Online]. Available: `https://www.humus.name/index.php?page=News&ID=228` (visited on 05/14/2024).

[30] M. Pharr and R. Fernando, *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*. Addison-Wesley Professional, 2005.