Performance Comparison of Meshlet Generation Strategies

Mark Bo Jensen
Technical University of Denmark

Jeppe Revall Frisvad
Technical University of Denmark

J. Andreas Bærentzen
Technical University of Denmark

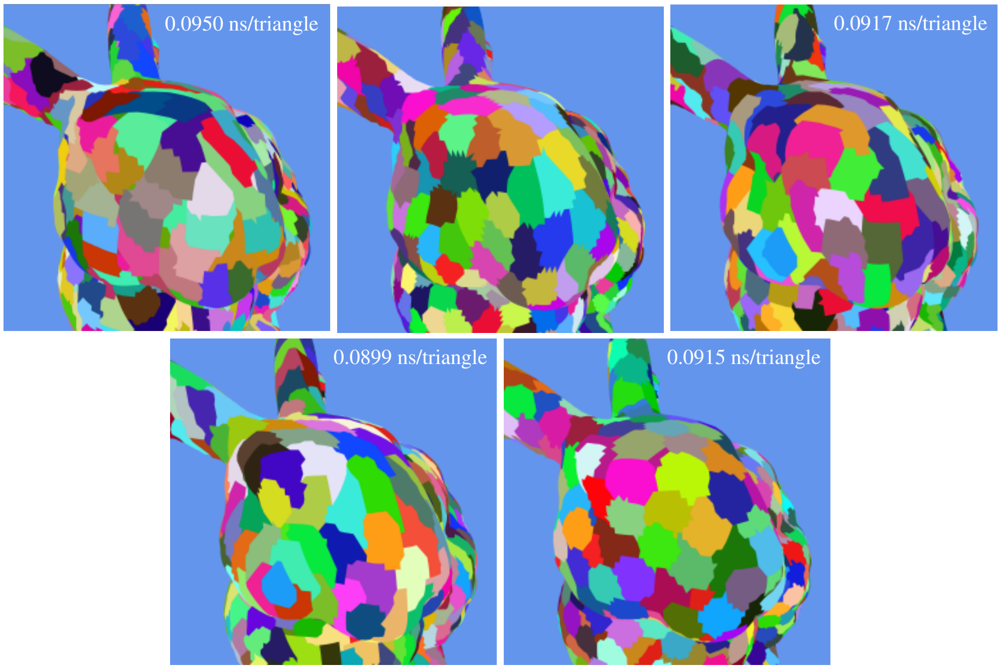

Figure 1. Different methods for organizing the triangles of the Stanford Bunny into meshlets. Each colored patch is a meshlet. From top left to bottom right: NVIDIA [Kubisch 2018b] with an optimized index buffer, k-medoids [Kaufman and Rousseeuw 1990], greedy (ours), bounding sphere (ours), and Kapoulkine [2017]. We describe the details of the methods in Section 3. Each image shows the render time in nanoseconds per triangle. The time is based on a linear regression fitted to the render time of six meshes as a function of their triangle count. Because k-medoids has too few data points, we omit its time.

Abstract

Mesh shaders were recently introduced for faster rendering of triangle meshes. Instead of pushing each individual triangle through the rasterization pipeline, we can create triangle clusters called meshlets and perform per-cluster culling operations. This is a great opportunity to efficiently render very large meshes. However, the performance of mesh shaders depends on how we create the meshlets. We tested rendering performance, on NVIDIA hardware, after the use of different methods for organizing triangle meshes into meshlets. To measure the performance of a method, we rendered meshes of different complexity from many randomly selected views and measured the render time per triangle. Based on our findings, we suggest guidelines for creation of meshlets. Using our guidelines we propose two simple methods for generating meshlets that result in good rendering performance, when combined with hardware manufactures best practices. Our objective is to make it easier for the graphics practitioner to organize a triangle mesh into high-performance meshlets. To support this we have uploaded our code to https://github.com/Senbyo/meshletmaker.

1. Introduction

Rasterization is fast and highly parallelized on the graphics processing unit (GPU). In extended reality (xR) applications, where too low a frame rate breaks the immersion and potentially causes motion sickness [Rebenitsch and Owen 2016], rasterization is the method of choice. Rasterization is however triangle bound, which means that every triangle must be processed for every frame. This can be prohibitively expensive if we want to visualize massive triangle meshes in xR applications. Equally, it is especially in xR applications that we need massive triangle meshes, because the user is free to closely inspect the geometry from arbitrary points of view.

To facilitate a higher triangle throughput, which helps uphold high frame rates even for massive meshes, the rasterization pipeline was recently modified to enable clustering of triangles into meshlets [Kubisch 2018a; Kubisch 2020]. Meshlets improve performance by enabling us to process and cull geometry at a coarser level of granularity than triangles [Jensen et al. 2021]. This relaxes the triangle boundedness, because the pipeline no longer needs to process all the triangles that are submitted to it. This modified pipeline is called the mesh shading pipeline.

Mesh shading is now directly exposed in Vulkan, DirectX 12, and OpenGL [Kubisch 2018a]. This gives rise to the question of how to best create the meshlets, i.e., the triangle clusters. Some developers, notably Kapoulkine [2017] and NVIDIA [Kubisch 2018b], have released code for organizing triangle meshes into meshlets, but the question of how to form meshlets that deliver good rendering performance has received limited attention. In this paper, we evaluate the rendering performance when using different approaches for organizing triangle meshes into meshlets. Our tests include six different meshes consisting of 70 thousand to 39 million triangles. We evaluated performance by rendering the meshes from many randomly selected views while measuring render time. The tests were carried out on NVIDIA hardware, using NVIDIA's best practices for meshlet sizes. These may differ between hardware manufactures, which is important to keep in mind when generalizing our results to other GPUs. To our surprise, we found that meshlet collections produce lower render times when using local and greedy algorithms.

We also conducted a small explorative study into different meshlet descriptors in order to investigate how they affect render times. A meshlet descriptor is a small structure that keeps track of the meta data surrounding a meshlet. Apart from describing different algorithms for forming meshlet collections and reporting their rendering times, our main contribution is to identify the most important metrics to consider when assessing the quality of a meshlet collection.

1.1. Related Work

The GPU was originally introduced as special purpose hardware for triangle rasterization. Over the past few decades, GPUs have evolved into highly efficient and very general architectures for parallel computation [Haines 2006; Dally et al. 2021]. The GPU is connected to the rest of the computer via a PCI-express bus, which is used for transferring data from main memory. The bandwidth of this bus can become a bottleneck [Hoppe 1999] when working with large datasets, such as very big triangle meshes. To mitigate this issue, one can use mesh representations that minimize the data footprint such as *triangle strips*. A triangle strip is a sequence of triangles in which adjacent triangles share an edge. Each triangle is represented by three points in space, called vertices. As the GPU processes each vertex, it is kept in memory as long as it is being used. This is exploited by the triangle strip since adjacent triangles share two out of three vertices, meaning that rasterizing the next triangle in the strip only requires processing one more vertex. Each vertex is generally used by more that two triangles, which means that the vertices will have to be present at different places in the triangle strip. Instead, an index buffer can be used to represent the triangle strip. This is then filled with indices that can be used to offset into a vertex buffer, avoiding vertex duplication.

To organize a mesh into triangle strips, we need a path through the mesh where each triangle is only visited once. This is equivalent to finding a Hamiltonian circle in the dual graph of the mesh, which is an NP-complete problem [Dillencourt 1996]. As a result, greedy approaches for creating triangle strips have been explored instead. Arkin et al. [1996] generated triangle strips by greedily adding triangles with fewest adjacent triangles to the strip. This approach avoids leaving behind isolated triangles (triangle islands). The algorithm is made for a graphics API that predates OpenGL, called Iris GL. Iris GL has a command that makes it possible to change the vertex order of the last processed triangle, which makes it possible to change the direction of a triangle strip. Since OpenGL does not have this command, degenerate triangles are added to the triangle strip in order to stitch strips together, at the cost of one extra vertex. Evans et al. [1996] sought to minimize this use of degenerate triangles by generating triangle strips based on a global heuristic that looks for large patches that can easily be converted into large strips.

The *generalized triangle mesh* introduced by Deering [1995] relies on a special-purpose hardware-accelerated cache called the mesh buffer. This buffer stores vertices through explicit commands. Using a mesh buffer makes it possible to exploit that vertices are on average connected to six triangles, which is hard to fully utilize with triangle strips [Deering and Nelson 1993]. Chow [1997] introduced an algorithm for converting meshes into generalized triangle meshes.

Hoppe [1999] relied on the post transform and lighting cache (post-T&L cache). The post-T&L cache is part of the vertex shading pipeline. The vertex shading pipeline is the traditional rasterization pipeline that is used to process geometric data and turn it into rasterized images. The post-T&L cache holds the most recently processed vertices that have not yet been converted into primitives. Using this, Hoppe optimized triangle strips by reusing the vertices in the cache as much as possible. Several others have built on this principle to further improve rendering performance [Lin and Yu 2006; Forsyth 2006; Sander et al. 2007]. In 2006, with the introduction of the unified shader model [Lindholm et al. 2008], the GPU became massively parallel, allowing for all shader stages to be run on all the generic processors on the GPU. This led Kerbl et al. [2018] to question whether the post-T&L cache is still a part of the GPU architecture. Based on empirical evidence obtained through vertex shader invocations, they showed that modern GPUs turn the index buffer into smaller batches and process these in parallel.

A new rasterization pipeline called the *mesh shading pipeline* was introduced with NVIDIA's Turing architecture [Kubisch 2018a]. This pipeline lets the GPU process the mesh in small clusters of triangles, aptly named meshlets, instead of individual triangles. The pipeline no longer has the fixed function batching that Kerbl et al. [2018] found in the vertex shading pipeline. Instead, this is done by the programmer, providing the opportunity to make more-informed decisions on how the mesh is batched into meshlets. Since each meshlet is processed in parallel, there is no longer a post-T&L cache to hold processed vertices; instead each processor has a cache of shared memory that all the threads on that processor can access. Since the batching is done before rendering, it does not need to take place again every time a new frame is rendered, removing some overhead. The pipeline expects a local index buffer for each meshlet as an output from the mesh shader stage, so this can either be precomputed or generated in the mesh shader. An optional task shader stage can run before the mesh shader to control culling, tessellation, and other things before it dispatches meshlets. The fragment shader stage is unchanged. Kubisch [2018a] provided an excellent overview of the hardware limits, built-in variables, and recommendations for the mesh shading pipeline.

The mesh shading pipeline is an evolution of the vertex shading pipeline. The implicit triangle batching in the vertex shading pipeline is made explicit. Instead of the global post-T&L cache, each streaming multiprocessor can utilize shared memory for vertex reuse. This means that both pipelines benefit from mesh optimizations that increase spatial locality of triangles, and this could arguably be why the mesh shading pipeline has received surprisingly little academic attention. Instead, academic attentions has focused on real-time ray tracing and methods based on machine learning. These have become hardware accelerated, leading to a more diverse set of viable methods for efficient, high-quality graphics.

Wihlidal [2016] showed how the graphics pipeline can benefit from the clustering of triangles and compute-based culling of these clusters. Jensen et al. [2021] showed that the mesh shading pipeline has great potential for visualizing large geometric datasets, and Unterguggenberger et al. [2021] showed how the mesh shading pipeline can be used for dynamic meshes. Mesh shaders work well for rendering large terrain [Santerre et al. 2020] and can be used for continuous level of detail [Englert 2020]. In the gaming industry the mesh shading pipeline has been adopted and is now part of Unreal 5's virtualized geometry pipeline called Nanite [Karis et al. 2021]. It is also possible to find GitHub repositories with mesh processing tools for the mesh shading pipeline [Walbourn 2014; Kapoulkine 2017; Lempiainen 2020]. Neff et al. [2022] investigated texture atlases to reduce meshlet overdraw. In this paper, we explore different clustering strategies for meshlet generation and distill two key principles that lead to better real-time rendering performance when generating meshlets.

2. Meshlets Descriptors

The buffer setup that we use with the mesh shading pipeline has three buffers; see Figure 2. A local index buffer is divided into one section for each meshlet, and the local indices start from 0 in each section. The indices are all 8-bit because they refer to the local indices within a single meshlet. The hardware limit for vertices in a single meshlet is 256, so 8 bits suffice. The global index buffer is also divided into sections, one for each meshlet. This buffer differs from the traditional index buffer in the sense that index duplication is reduced. If one meshlet uses a vertex several times, the local index that points to the same global index is duplicated instead. The last buffer is simply the vertex buffer, which is the same as the one used for the vertex shading pipeline. With these buffers, all we need is a small descriptor for each meshlet providing information about it for the multiprocessor. NVIDIA suggests keeping the size of the meshlet descriptors to 128 bits, which, on their hardware, is equivalent

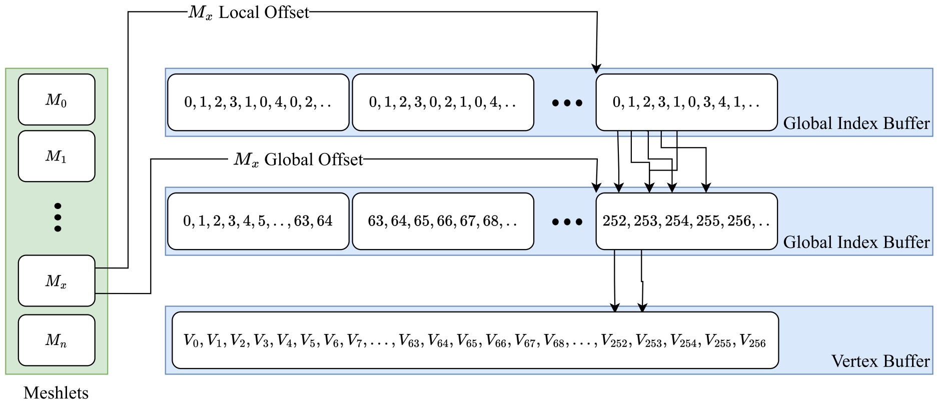

Figure 2. The three buffers used by the GPU when processing meshlets: local index buffer, global index buffer, and vertex buffer. The meshlet descriptor has offsets into these buffers. Note that global indices (re)appear in all meshlets in which they are used.

to the minimum amount of data that is fetched on a GPU-side load instruction. The meshlet descriptor is a small structure that keeps track of the meta data surrounding a meshlet. It needs to at least hold offsets into the global and local index buffers, as well as the number of primitives and vertices used in the meshlet. Other than this, the descriptor can also store a bounding box, an average normal for the meshlet, or any other information that the programmer wants to have associated with a meshlet.

The layouts of four different descriptors are in Tables 1 and 2. All descriptors use at most 128 bits. All descriptors pack a bounding box into 48 bits, namely 8 bits for the minimum and maximum coordinate on the x-, y- and z-axes. The bounding box coordinates are relative to the extent of the mesh bounding box. They all use 8 bits for describing the number of primitives and vertices in the meshlet. The normal cone is represented by a normal and an angle packed into 24 bits. The normal and cone angles are mapped into octants based on Cigolle et al. [2014]. All data in a descriptor is packed into four 32-bit unsigned integers. The NVIDIA descriptor A packs the 8-bit cone angle partially into two 32-bit unsigned integers: the four upper bits in one and the four lower bits in the other. The remaining three descriptors pack the 8-bit cone angle together, which saves some unpacking within the mesh shader. The biggest point of divergence between the four descriptors lies in how they store the offsets required for the global and local index buffers.

The NVIDIA descriptor A has 20 bits left for indexing into both the local and the global index buffer. This means that meshes that require an offset that is larger than 2^{20} will need to be broken into several draw calls.

The NVIDIA descriptor B takes these same 40 bits and uses 32 of them for offsetting, which allows for much larger meshes. The downside of this is that the offsets

NVIDIA Descriptor A	
	Bits
Bounding box	48
No. vertices	8
No. primitives	8
Global idx offset	20
Local idx offset	20
Normal cone	24

NVIDIA Descriptor B	
	Bits
Bounding box	48
No. vertices	8
No. primitives	8
vertexPack	8
Index buffer offset	32
Normal cone	24

Table 1. The memory layout of two meshlet descriptors proposed by NVIDIA [Kubisch 2018b]. Meshlet descriptors are 128-bit data structures that are used in task and mesh shaders.

Task Shader Meshlet Descriptor	
	Bits
Bounding box	48
Normal cone	24

Mesh Shader Meshlet Descriptor	
	Bits
No. vertices	8
No. primitives	8
Global idx offset	32
Local idx offset	32

Table 2. A descriptor for the task shader stage (left) and another descriptor for the mesh shader stage (right). Use of different descriptors for task and mesh shaders is an alternative to using the same descriptor for both shaders.

into the global and local index buffers need to be aligned, as the same offset is used in both buffers. The remaining 8 bits are used to describe how the global indices are packed, i.e., if they are 16-bit or 32-bit numbers. This effectively means that the global indices can be packed into 16 bits for meshlets that only use global indices smaller than 2^{16}.

The third descriptor separates the task and meshlet descriptors, meaning that it uses 256 bits for each meshlet instead of 128. But it only loads 128 bits per shader stage. By doing that, we can get rid of the task shader–related data in the mesh shader descriptor and vice versa. That way we can allow 32 bits for both the global and local index buffer offsets. So here we require no alignment between the buffers. We refer to this as the split descriptor.

Figure 3 shows an alternative buffer setup for a monolithic meshlet descriptor. The monolithic descriptor is also divided into two descriptors, to allow for 2×32 bits offsetting. One offsets into the local index buffer, and instead of using a global index buffer, the second offsets directly into the vertex buffer, which is divided into sections for each meshlet. On the one hand, the trade-off here is memory, since some vertices will be duplicated and appear in several sections. On the other hand, no global index buffer is needed. The duplication is required for all vertices that live on the border of a meshlet. So, the four different descriptors all come with different memory footprints

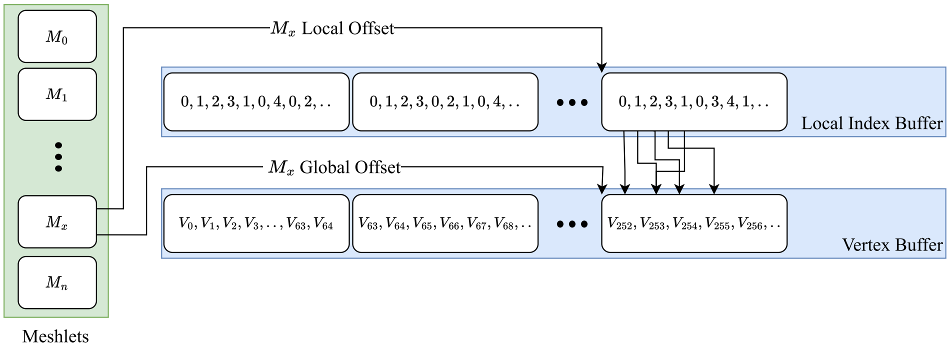

Figure 3. A monolithic version of the buffer setup used by the GPU when processing meshlets, using only two buffers: local index buffer and vertex buffer. The monolithic meshlet descriptor has offsets into these. Note that vertices (re)appear in all meshlets in which they are used.

as well as some variations in how much GPU-side unpacking they require.

Each meshlet can only contain a certain number of vertices and primitives. These numbers dependent on the GPU hardware. In the case of NVIDIA's 2000 RTX series, the hardware limits are 256 vertices and 256 primitives. Lower values can be set as well. NVIDIA suggests using either 32 or 64 vertices and 40, 84, or 126 primitives for each meshlet. In this paper, we use 64 vertices and 126 primitives throughout, which is the same as NVIDIA used in their meshlet sample [Kubisch 2018b].

We used NVIDIA descriptor B when comparing the rendering performance of different meshlet generation methods because it allows us to process large meshes with one draw call. For our descriptor comparison, we compared all four descriptors while using the meshlet clustering method with best performance.

3. Meshlet Clustering Methods

The following paragraphs describe the different methods for organizing a mesh into meshlet collections (clusters) that we compare. Figure 1 exemplifies the differences between the meshlets generated by the different methods.

NVIDIA On behalf of NVIDIA, Kubisch [2018b] provided an example of organizing a mesh into meshlets. The meshlets are created one at the time by going through the index buffer. New primitives and vertices are added to the current meshlets as long as there is room for more. When it is full, a new meshlet is created. This process is repeated until the algorithm has gone through the entire index buffer. Every time a primitive is added to a meshlet, it generates local 8-bit indices for the vertices, or reuses existing local indices if the vertices are already in the meshlet. It de-duplicates the global vertex indices, meaning that the global index of a vertex is only stored once,

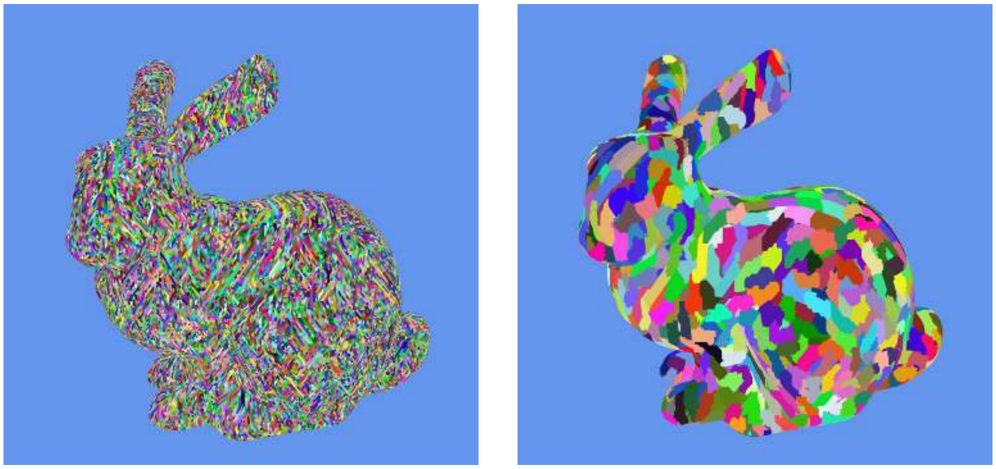

Figure 4. A visualization of the NVIDIA-generated meshlets without optimizing the index buffer (left) and after optimizing the index buffer using the tipsify algorithm (right).

Method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA	0.13	0.23	0.83	1.35	3.88	3.81
NVIDIA+tipsify	0.12	0.22	0.72	1.20	3.62	3.33

Table 3. Average render times in milliseconds.

in each meshlet that uses the vertex, instead of being stored once for each triangle of which it is a part. Instead, the local indices are stored for each triangle. Because the local index buffer is 8-bit and the global index buffer is 16- or 32-bit, this saves space. The approach has a dependency on the original connectivity of the index buffer, and the resulting number of meshlets, as well as the vertex reuse within the meshlets, is highly dependant on the structure of the index buffer. Figure 4 compares the resulting meshlets for the Stanford Bunny when using an unoptimized index buffer versus an index buffer that is optimized. We implemented the tipsify algorithm from Sander et al. [2007] to optimize the locality in the index buffer. Table 3 shows the difference in render time for the different meshes with and without optimization of the index buffer before running NVIDIA's meshlet clustering algorithm. The algorithm requires tuning of the cache size as different cache sizes result in different index buffers, which in turn affect the size of the meshlet collections. We explored different cache sizes and achieved the best result using 25 for Bunny, 26 for Happy, 24 for Dragon and Skull, and 20 for Nobby and Wing. Throughout the rest of the paper, we will be using the NVIDIA algorithm with a tipsified index buffer and refer to it as NVIDIA+tipsify.

Kapoulkine Arseny Kapoulkine [2017] maintains a widely used and popular library called meshOptimizer. The library has several functions that improve, pack, and optimize meshes for better render performance, and it includes a meshlet generation strategy. First, the library creates a data structure based on triangle and vertex adjacency. A centroid and a normal are then calculated for each triangle, and the area of the mesh is also calculated. The area is used to create an expected meshlet area, assuming square flat patches. In addition, a kd tree is created from the triangle structure. All this is used to create the meshlets. The kd tree is used to pick the starting triangle for a meshlet, and the adjacency structure is then used to look up the nearest triangles. Each triangle gets two ratings: one based on vertex reuse and another based on how much it increases the area of the meshlet. Regarding triangle reuse, triangles that already have vertices in the meshlet get a higher rating. Triangles islands also get higher importance. Should it happen that there is room for more triangles in the meshlet but none available on the border, the algorithm uses the kd tree to look up the nearest available triangle. The meshlet generation algorithm allows one to set a weight for the triangle normals, which will make it weigh these more when picking the next triangle for the current meshlet. We set it to 0.0, 0.5, and 1.0, and we found that 0.0 produced the best results for the large meshes while the difference between the weights only had a very small impact on the small meshes. Because of this, we report our results with the weight set to 0.0.

Greedy We have developed a greedy algorithm that uses a list of vertices, where each vertex contains information about which triangles it is part of. The algorithm takes the first vertex and then, from that, grows out the triangle cluster until a meshlet is full. If a meshlet hits the vertex maximum before the primitive maximum, we look at the border of the meshlet for triangles that already have all vertices in the meshlet, and add these. A new meshlet is then started from a vertex on the border of the meshlet that was just completed, and the process is repeated. If a meshlet runs out of available triangles on its border, we go back to the list and pick the next available one. Because of this, the algorithm is sensitive to the order of the vertex list. We therefore use a heuristic to sort the list before running the algorithm. We find that half the time sorting according to the biggest bounding box axis length gives the best result. In particular, this is the case for the three biggest meshes. We also developed a version using a triangle list instead of a vertex list, but found that the vertex-based algorithm always outperformed the triangle-based one. This is most likely because the meshlet border for vertices is based on all the triangles that the vertices in the meshlet touch, while the border in the triangle version is based on all triangles that share an edge with triangles that are already in the meshlet. This effectively means that the border is “larger” for the vertex version, which results in fewer meshlets overall. Moving forward we only report on the vertex-based algorithms and use the heuristic of sorting the vertex list based on the longest bounding box axis of the mesh, from low to high. The pseudo-code for our greedy algorithm is in Listing 1.

Listing 1: Greedy algorithm.

```
input  : Sorted VertexList, vertex, and triangle data structure
output : Meshlet collection
for vertex in vertexList do
    if vertex is used then continue
    queue.push(vertex)
    while queue is not empty do
        curVertex $\leftarrow$ queue.front()
        queue.pop()
        for triangle in curVertex.triangles do
            if triangle is used then continue
            for vert in triangle do
                if vert is not used then queue.push(vert)
            if Meshlet is full then
                if Meshlet.triangles $< 124$ then
                    for triangle in Meshlet.border do
                        if triangle.vertices in Meshlet then Meshlet.add(triangle)
                queue $\leftarrow \{$curVertex$\}$
                Meshlet $\leftarrow$ new Meshlet
                break
            Meshlet.add(triangle)
```

Bounding sphere Our more advanced strategy is similar to the greedy one, except here we grow a bounding sphere around the starting vertex and use an algorithm by Bærentzen and Rotenberg [2021] to add triangles that minimize the radius of this bounding sphere. In addition to striving for a minimal bounding sphere radius, we also (inspired by Kapoulkine) prioritize triangles with vertices already in the meshlet and triangle islands. The pseudo-code for our bounding sphere algorithm is in Listing 2. The *vertexScore* and *newRadius* variables decide which triangle on the meshlet border to add next. The vertexScore for a triangle increases for each vertex that is already in the meshlet and if it is considered a triangle island. The newRadius of the bounding sphere is calculated for each triangle, and the triangle with the smallest increase is picked given that its vertexScore is not less than another triangle. If the triangle only has one vertex in the meshlet, it is not even considered.

k-medoids One way to create clusters of triangles is by turning a mesh into smaller partitions using k-medoids [Kaufman and Rousseeuw 1990]. Though this is an algorithm normally used for unsupervised learning, to investigate if and how many clusters a dataset might have, we use it to obtain balanced clusters. We chose the k-medoids approach because it works along the mesh surface, whereas the more commonly known k-means clustering would use a centroid, the cluster mean, to represent a cluster. A centroid detached from the surface, one the one hand, easily results in clusters with triangles that are not connected. A medoid, on the other hand, is an actual data point within the cluster that is most suited to represent that cluster. These can

Listing 2: Bounding sphere algorithm.

```
input  : Sorted vertexList, vertex, and triangle data structure
output : Meshlet collection
radius $\leftarrow$ 0
center $\leftarrow$ vec3(0)
for i $\leftarrow$ 0 to vertexList.size do
    vertex = vertexList[i]
    if vertex is used then continue
    bestTriangle $\leftarrow$ null
    newVertex $\leftarrow -1$
    newRadius $\leftarrow$ FLT_MAX
    BestNewRadius $\leftarrow$ FLT_MAX - 1
    vertexScore $\leftarrow$ 0
    bestVertexScore $\leftarrow$ 0
    for triangle in Meshlet.border do
        if triangle is used then continue
        for vert in triangle do
            if vert is in Meshlet then vertexScore += 1
            else newVertex $\leftarrow$ vert.index
        if vertexScore equals 3 then newRadius $\leftarrow$ radius
        else if vertexScore equals 1 then continue
        else newRadius $\leftarrow 0.5 \cdot (\text{radius} + \|\text{center} - \text{vertexList[newVertex]}\|)$
        trianglesInMeshlet $\leftarrow$ 0
        for tri in triangle.neighbours do
            if tri is in Meshlet then trianglesInMeshlet += 1
        if triangle.neighbours.size equals trianglesInMeshlet then vertexScore += 1
        if vertexScore $\geq$ bestVertexScore or newRadius $\leq$ bestNewRadius then
            bestVertexScore $\leftarrow$ vertexScore
            bestNewRadius $\leftarrow$ newRadius
            bestTriangle $\leftarrow$ triangle
    if bestTriangle == null then
        for triangle in vertex.neighbours do
            if triangle is used then continue
            bestTriangle $\leftarrow$ triangle
            center $\leftarrow \textbf{sum}(\text{bestTriangle.vertices})/3$
            bestNewRadius $\leftarrow \textbf{max}(\|\text{center} - \text{bestTriangle.vertices[0]}\|, \textbf{max}(\|\text{center} -$
              $\text{bestTriangle.vertices[1]}\|, \|\text{center} - \text{bestTriangle.vertices[2]}\|))$
    if bestTriangle equals null then
        i += 1
        continue
    radius $\leftarrow$ bestNewRadius
    center $\leftarrow \text{vertexList[newVertex]} + (\text{radius}/(\epsilon + \|\text{center} - \text{vertexList[newVertex]}\|)) \cdot$
      $(\text{center} - \text{vertexList[newVertex]})$
    if Meshlet is full then
        if Meshlet.triangles $<$ 124 then
            for triangle in Meshlet.border do
                if triangle.vertices are in Meshlet then Meshlet.add(triangle)
        continue
    Meshlet.add(triangle)
```

be found by minimizing the dissimilarity within a partition. The k-medoids method partitions the mesh into k clusters and finds the medoid for each of these clusters. The medoid is the triangle with the shortest distance to all other triangles in the cluster. The algorithm runs in two steps after creating an initial clustering of the mesh. First, the medoids of all clusters are found. All triangles are then compared to these medoids and assigned to the cluster with the most similar medoid. These two steps are repeated until convergence [Kaufman and Rousseeuw 1990]. The dissimilarity can be expressed through a distance metric between triangles. We run the algorithm on a triangle data structure, where the distance between two triangles is equal to the number of adjacent triangles we have to walk through to get from one to the other. The convergence criterion is to have an average distance close to zero between the new and old cluster centers, meaning that cluster centers moved very little in the last iteration. We start the algorithm with a number of clusters found by dividing the total number of triangles by the maximum number of triangles in a meshlet. After convergence we check if the clusters fit into meshlets. If not, then we add one new cluster and repeat. By only adding one new cluster we minimize the total number of clusters at the cost of longer processing times.

The five methods just mentioned vary quite a bit in implementation complexity. NVIDIA's algorithm is arguably the simplest to implement because it just directly works on the index buffer. After this comes the greedy algorithm that uses a triangle and vertex adjacency structure in a sorted list instead of the index buffer, with the bounding sphere version adding a little complexity in terms of a triangle scoring function. Then we have Kapoulkine's method, which requires both a triangle and vertex adjacency structure, a kd tree, and two scoring functions. Lastly, we have the k-medoids algorithm, which requires not only a triangle adjacency structure but also two iterative steps based on the breadth first algorithm and which, to even be applicable, needs to be optimized and parallelized. The five methods also have different processing times. The processing times for all the different methods across the different meshes can be seen in Table 4. The processing times are presented without the time it takes to load the obj file. As is evident from the table, the processing times of the k-medoids algorithm increase dramatically due to its runtime complexity being $O(n^2 + k^2)$ for n triangles and k clusters. Because of this, we decided to not use it on the largest meshes. The processing times for the NVIDIA algorithm include the preprocessing of the index buffer by the tipsify algorithm.

4. Experimental Setup

We compared the five different algorithms to see which one performs best, and why. Our hope is that this comparison allows us to distil more general principles for meshlet

Method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA+tipsify	13.54	222.56	1372.17	2933.59	5283.1	7458.69
Kapoulkine	46.861	297.767	5000.53	10679.4	16926.2	28805.1
Greedy	133.603	2181.37	14503.9	33335.7	66351.3	93513
Bounding sphere	255.081	3842.83	25211.2	55720.4	100415	154677
k-medoids	50769.2	1.245e7	4.553e8	N/A	N/A	N/A

Table 4. Processing times in milliseconds.

Bunny	Happy	Dragon	Skull	Nobby	Wing
V: 34 817	V: 543 652	V: 3 609 600	V: 7 252 445	V: 16 960 045	V: 19 473 581
T: 69 630	T: 1 087 716	T: 7 219 045	T: 14 504 882	T: 32 905 214	T: 38 629 758

Figure 5. The six meshes used in our experiment and their numbers of vertices (V) and triangles (T).

generation that transcend the specific hardware and numbers used. To make sure that no bias is introduced into the experimental process, we set up a Vulkan visualization engine, using Vulkan 1.2.176.1 with four MSAA samples per pixel, that visualizes all the objects from a new random point in space each frame. Our efforts to randomize the view point are to average out the effect of overdraw. By setting the random seed, we made sure that all algorithms were tested with the same sequence of view points; we did this for a total of 100,000 frames and recorded different statistics for each method. The frames were rendered at a resolution of 1280×720 pixels. Because the first frame includes data transfer to the GPU, it was discarded. The analysis was carried out on the subsequent 99,999 frames. All experiments were run on a desktop computer with an Intel Core i9-9900k, 64GB of DDR4-2666 RAM, and one NVIDIA GeForce RTX 2080 Ti Turbo OC with 11GB of GDDR6 RAM. The shader code used to process the meshlet is based on the NVIDIA GitHub repository showcasing the use of mesh shaders in Vulkan (https://github.com/nvprosamples/gl_vk_meshlet_cadscene). For our experiments, we turned off triangle culling in the mesh shader and only did frustum and backface culling in the task shader. We report our results in average render time per frame in milliseconds, while also exploring other metrics surrounding the meshlets that impact the render times. We used six different models for our tests in this paper. The vertex and triangle counts of each model are listed in Figure 5. The Stanford Bunny, Happy Buddha, and Asian Dragon are from the Stanford 3D Scanning Repository (https://graphics.stanford.edu/data/3Dscanrep/). The Seal Skull was 3D scanned into a point cloud and digitally reconstructed as a triangle mesh (https://www.morphosource.org/projects/000355763). The topology-optimized airplane wing [Aage et al. 2017; Aage et al. 2020] is the largest model in our com-

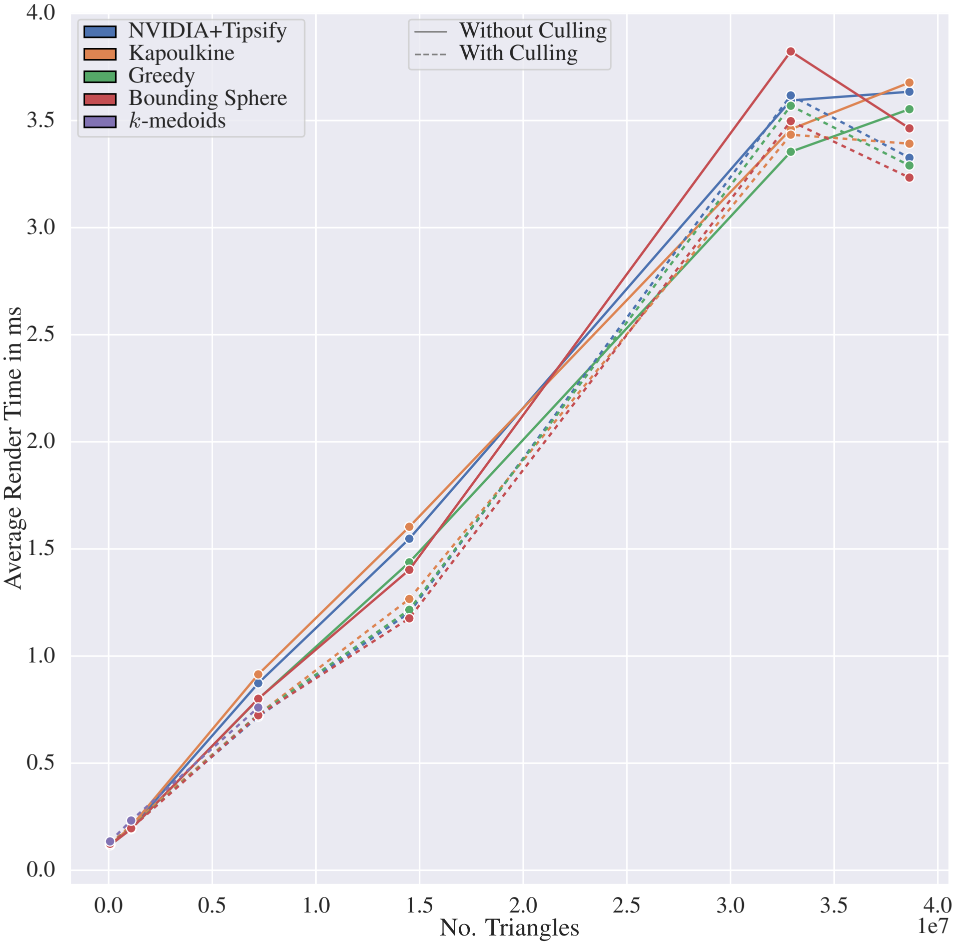

Figure 6. Average render time as a result of triangles based on the six meshes. Render times with meshlet culling are presented with a dashed line, and render times without culling are presented with an solid line.

parisons. The last mesh was created with PrusaSlicer (https://www.prusa3d.com/) using a model called Nobby (https://www.prusaprinters.org/prints/35338-nobby-octopus-sculpt). We used the same experimental setup when testing the different meshlet descriptors, using the best-performing meshlet generation algorithm.

5. Results

We are interested in finding a good clustering algorithm for meshlet generation. To investigate this, we plot the render times of the different algorithms as a function of triangle count in Figure 6. We see a fairly linear trend. The solid lines show render times without meshlet culling, while the dashed lines include meshlet culling. Fig-

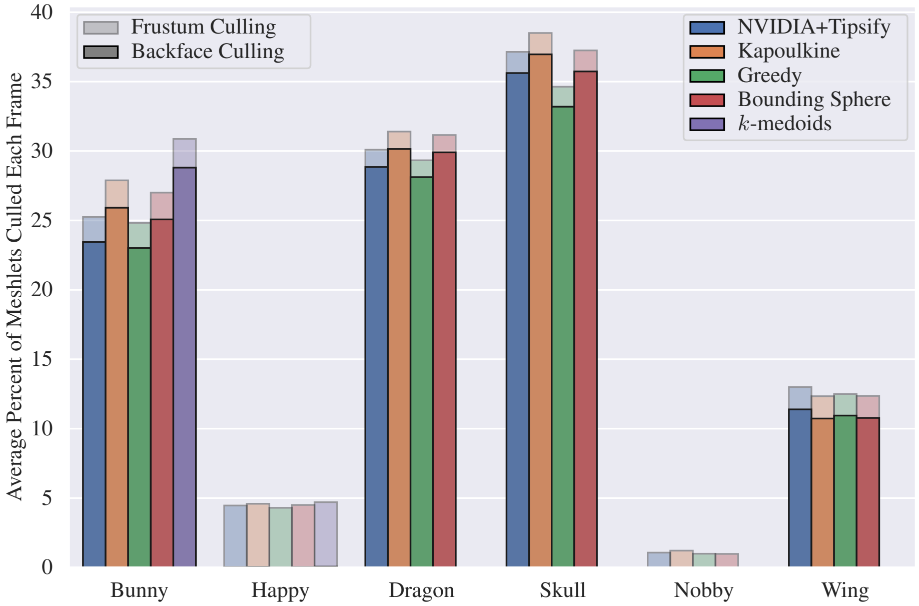

Figure 7. The average percent of meshlets that are culled for each frame when using the five different clustering algorithms. The culled meshlets are divided into two, the backface-culled meshlets are represented by the fully opaque bars, while the frustum-culled meshlets are represented by the semitransparent bars.

ure 7 shows what percentage of the meshlets were culled on average, each frame. The vertical axis shows the percentage of meshlets culled, and the opaque bars represent the number of meshlets that were backface-culled, while the semitransparent bars show the frustum-culled meshlets. The two bars are stacked on top of each other. From this plot, we see that for Nobby and Happy we had no backface culling at all. This is because the meshlets generated for these two meshes did not have well-defined average normals, and without a well-defined average normal, the meshlets cannot be backface-culled. For the Happy Buddha model, the reason is the roughness of the mesh surface—see Figure 8 (right)—and for Nobby, the reason is the tube structure used to mimic how a 3D print of the model would look—see Figure 8 (left).

The actual render times are listed in Table 5. Here, it is evident that the three smallest meshes exhibit no real difference in performance between the best-performing algorithms, but for the larger models, we see a clear difference in performance. Given the linear trend, we also fit a regression line to each algorithm and report the resulting slope in Table 6. The slopes are reported in nanoseconds per triangle, with and without culling, and we consider these slopes an overall measure of the performance of the different methods. The k-medoids method is omitted in this table due to too few data points. The smaller the slope, the less an algorithm grows in render time

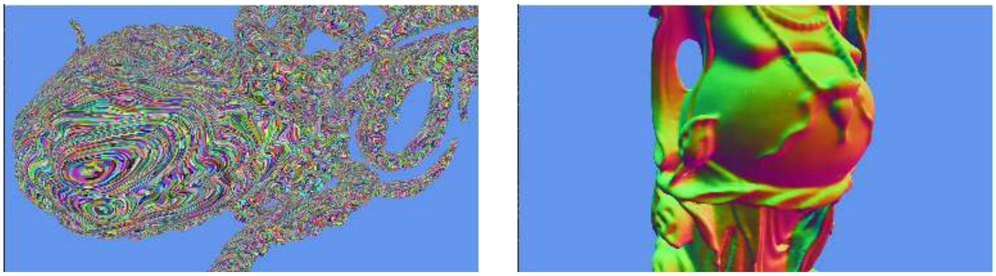

Figure 8. Visualizations of the long cylindrical meshlets of Nobby (left) and the normals of Happy Buddha illustrating its surface roughness (right).

Method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA+tipsify	0.12	0.22	0.72	1.20	3.62	3.33
Kapoulkine	0.12	0.21	0.73	1.27	**3.43**	3.39
Greedy	0.12	0.20	0.73	1.22	3.57	3.29
Bounding sphere	0.12	0.20	0.72	**1.18**	3.50	**3.23**
k-medoids	0.13	0.26	0.76	N/A	N/A	N/A

Table 5. Render times.

as more triangles are rendered. The bounding sphere algorithm achieves the smallest slope, meaning that when applied to our six meshes, it increased the least in render time as the number of triangles grew. Since the difference between the algorithms is evident both with and without culling of meshlets, it means that the clustering within the meshlets themselves also contributes to the difference in render times. The slope from the linear fit is only based on six meshes. These six meshes however are all quite different, and as such do a good job of covering the input space of different models. This results in a fairly decent fit, especially when looking at the trend for the algorithms without culling, but it should also be noted that with six meshes it is hard to extrapolate to new meshes that might not follow the trends presented here. When we compare the render times to the implementation complexity of the algorithm, we have that NVIDIA's algorithm is the simplest to implement, but this comes with a performance hit. Alternatively we have Kapoulkine's algorithm that achieved good render times but is rather complicated to implement. Right in the middle we have the greedy algorithm. This has the second smallest slope while also being quite simple to implement.

Each meshlet has a maximum number of vertices and a maximum number of primitives that it can contain. We found that all methods (except k-medoids) have a very high average vertex count. For each meshlet collection, we found the average vertex fill (ratio of vertices in a meshlet to the maximum number it can hold). All other collections have an average above 0.99 (except for k-medoids with Bunny: 0.812,

Method	Without Culling	With Culling
NVIDIA+tipsify	0.0967	0.0930
Kapoulkine	0.0953	0.0915
Greedy	0.0929	0.0917
Bounding sphere	0.0974	0.0899

Table 6. The slope of a linear regression fitted to the six mesh render times based on the four different algorithms with and without culling. The slope shows how much an algorithm increases in render time as more triangles are rendered. The time is given in nanoseconds.

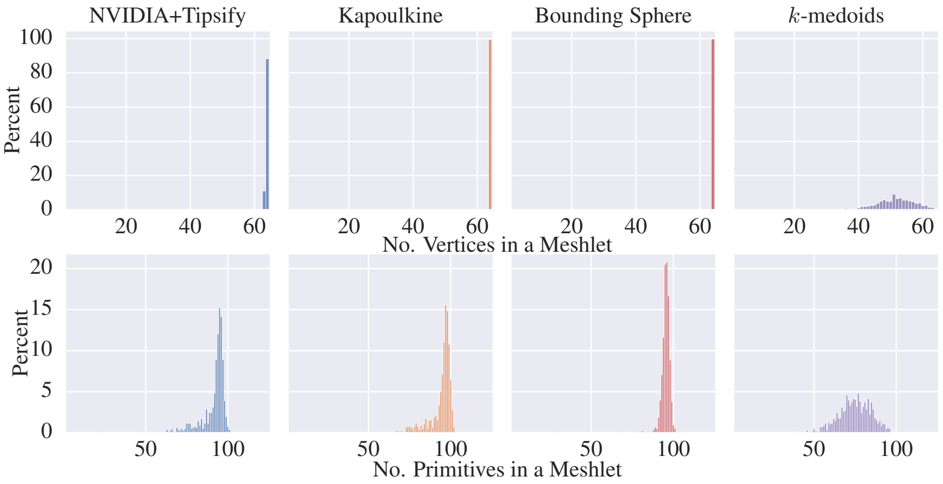

Figure 9. The distribution of the number of vertices and the number of triangles in each meshlet across four meshlet generation algorithms. The top row shows the vertices and bottom row is triangles. The meshlet collections are based on the Stanford Bunny mesh.

Happy: 0.770, and Dragon: 0.811). With all algorithms achieving close to vertex-complete meshlets, i.e., meshlets that are filled with vertices to the limit, the vertex completeness does not help us explain the differences in render times.

To see why k-medoids generates meshlet collections with a lower average vertex completeness, we compare its distributions to the other algorithms in Figure 9. Since the nature of the k-medoids algorithm is to balance out the clusters, we get a distribution of the number of vertices with two fat tails. This means that we will always be below capacity, and when we compare it to NVIDIA's, and especially Kapoulkine's, method, we see high peaks and only a tail to one side. Kapoulkine's algorithm performs better than both NVIDIA's and the k-medoids, and produces quite few meshlets when compared to the other two. The numbers of meshlets produced by the different methods for the different meshes are listed in Table 7. Since the k-medoids algorithm

Method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA+tipsify	762	12 419	77 998	163 565	287 171	429 192
Kapoulkine	740	11 257	76 127	152 261	288 230	438 582
Greedy	756	12 231	77 538	160 436	286 956	424 325
Bounding sphere	731	11 605	75 457	152 501	286 767	408 302
k-medoids	921	15 210	96 111	N/A	N/A	N/A

Table 7. Number of meshlets.

Method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA+tipsify	0.725	0.695	0.735	0.704	0.909	0.714
Kapoulkine	0.747	0.767	0.753	0.756	0.906	0.699
Greedy	0.731	0.706	0.739	0.718	0.910	0.723
Bounding sphere	0.756	0.744	0.759	0.755	0.911	0.751
k-medoids	0.600	0.568	0.596	N/A	N/A	N/A

Table 8. The average primitive fill for meshlet collections.

is trying to distribute the triangles and not the vertices, the distribution of the number of triangles shows the same two-tailed distribution. NVIDIA's and Kapoulkine's distributions are more interesting. Kapoulkine's has a peak at a high number of triangles and a tail that falls off toward smaller numbers, while NVIDIA's is the opposite. This is most likely because of the index buffer and how it does not promote locality as well as Kapoulkine's adjacency-based method, resulting in less locality and more unique vertices. These results informed us that greedy strategies ensure more vertex- and triangle-complete meshlets.

Since vertex completeness did not help differentiate the algorithms, we instead inspected triangle completeness. Table 8 shows the average primitive fill (ratio of primitives to the maximum number of primitives). Unlike the vertex count, the primitive count varies quite a bit more across the different algorithms and meshes. If we compare this to Table 5, we see a correlation between the methods that perform the best and their primitive fill being high (although not as simple as saying that the highest primitive fill yields the best render time). The primitive fill number also explains the variance in the meshlet collection sizes. If we look at NVIDIA's algorithm for instance, it produces more meshlets than the other algorithms. Since each meshlet holds fewer primitives, we need more meshlets to represent the meshes. The k-medoids algorithm does not achieve a high primitive fill for any of its three meshes. Since it fails to produce high vertex fill, it becomes even more difficult to achieve a high primitive fill. NVIDIA's algorithm has the lowest primitive fill and also performs the worst, which indicates that it is difficult to build meshlets directly from the index buffer.

The NVIDIA and k-medoids algorithms both generate meshlet collections with a somewhat wide distribution of vertices and primitives (Figure 9). To investigate how

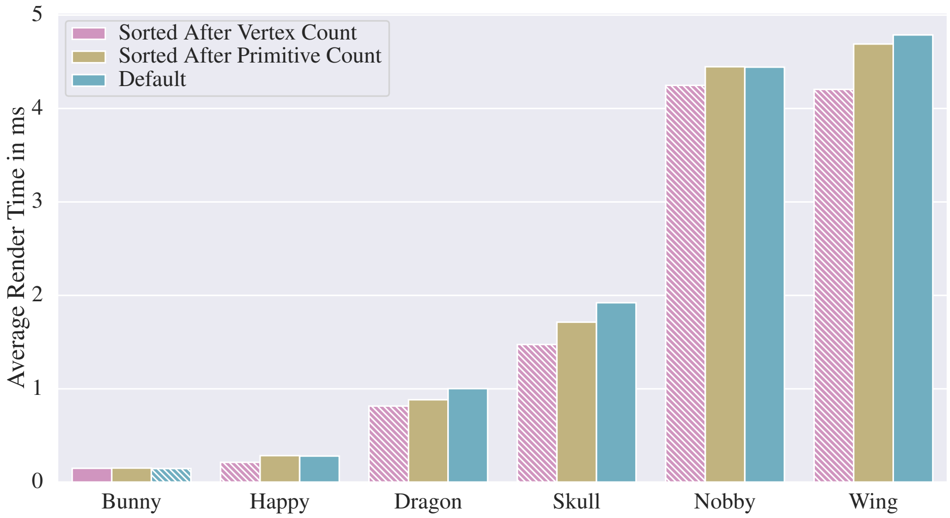

Figure 10. The average render times for the NVIDIA+tipsify meshlet collections for each mesh as a result of sorting the meshlet list that is send to the GPU. The list is sorted based on number of vertices and primitives. The resulting render times are compared to sending the meshlet list as is. The hatched bar for each mesh show the best performing ordering.

this impacts the performance of meshlet collections, we sort the meshlets with respect to the number of vertices and number of primitives. We only do this for the NVIDIA-based meshlet collections. As seen in Figure 10, the order of the meshlets does play a role. We clearly see that sorting after primitive fill yields the best results. This is most likely due to the fact that the average vertex completeness of the meshlet collections is above 0.9, and so sorting after primitives results in a more uniform load across the GPU. The reason why the render times are affected is that the GPU resources are used better. Meshlets are dispatched in groups to be processed in parallel, and if these groups are done processing at the same time, a new group can be dispatched without idle time. If the meshlets are of varying sizes, some will finish before others and will end up having to wait for the biggest meshlet to finish processing before a new group can be dispatched.

Since cullability increases performance of the meshlet collections, we find it interesting to explore the importance of the cullability of the meshlets. To test this we tweaked our bounding sphere technique for generating meshlets. When a meshlet runs out of new triangles to add from its border, we finish the meshlet instead of going back to the vertex list to look for new candidates. This enforces spatially coherent meshlets. By doing this we created more compact meshlets, making them more likely to be frustum-culled. This also reduces the chance of adding a triangle with a normal that deviates too much from the meshlet normal. The increased cullability comes at

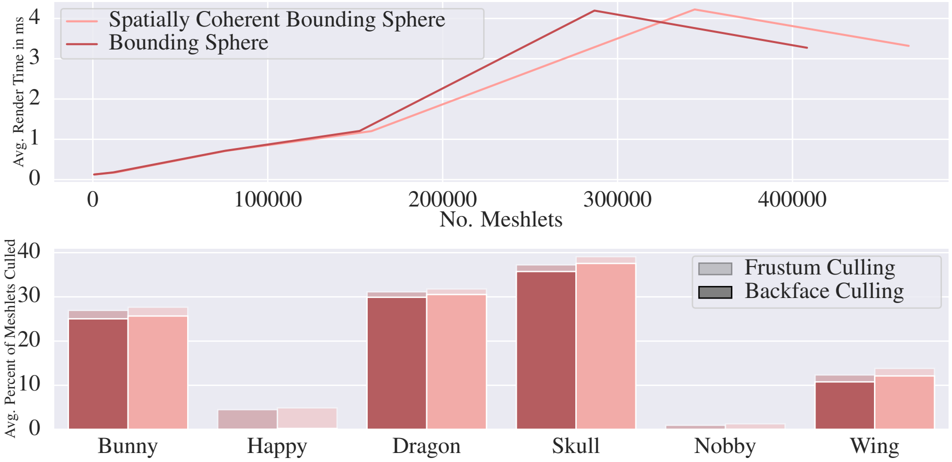

Figure 11. Comparison between the bounding sphere vertex meshlet collections with and without spatially coherent meshlets.The top plot shows the average render time, as a function of the size of the meshlet collections. The bottom plot shows the average percent of meshlets that are being culled per frame for each method.

the cost of a larger meshlet collection. In Figure 11, we see that the more-cullable spatially coherent meshlet collections are offset to the right of the normal meshlet collection because they contain more meshlets. For smaller meshes, the spatially coherent meshlet collections show better performance, despite having more meshlets. The increased number of meshlets seems to be offset by the larger amount of culling. The increased culling is however not sufficient to hide the larger loading and processing times for the big meshes. Here, the difference in render times between the two meshlet collections is small.

Meshlet Descriptor Comparison We used our bounding sphere algorithm to test the four different meshlet descriptors described in Section 2. The results are shown in Figure 12. The type of descriptor that hasd the best performance varies from mesh to mesh. We see the biggest difference in render times for Nobby. Here, the monolithic meshlet descriptor setup outperforms the other descriptors. The Nobby model is a representation of a 3D print; because of this, it consists of tubes. These tubes will have normals that point in all directions, making it impossible to form meshlets with well-defined normal cones, meaning that no or very little backface culling is taking place. Because of this, all visible meshlets are processed, which gives an interesting insight into how much the meshlet culling affects performance. The high average render time for the NVIDIA descriptor A is most likely a result of overdraw, because the mesh has almost no cullable meshlets.

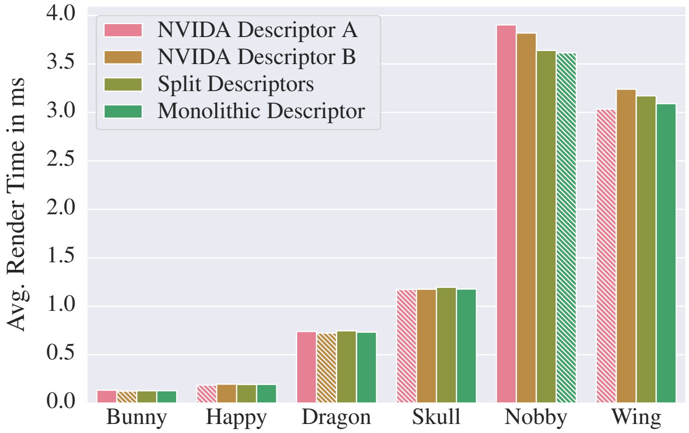

Figure 12. Performance comparison across the six meshes for the four different meshlet descriptors described in Section 2. For each mesh, the hatched bars highlight the descriptor with best performance.

6. Discussion

Most of our experiments show that vertex completeness is important. Exploring the meshlets generated from k-medoids displays this the best. The distributions from Figure 9 and render times from Table 5 express that one should prioritize vertex-complete meshlets over balanced meshlets. Our investigation into spatially coherent meshlets shows the same, albeit with a weaker signal. Spatially coherent meshlets result in better-cullable meshlets at the cost of generating more meshlets. Generating more meshlets means having a bigger distribution of vertices and primitives. The differences here are small when compared to the k-medoids results because the portion of meshlets with lower vertex completeness is small, but for bigger meshes it starts to affect performance more. More vertex-complete meshlets also mean more uniform meshlets, and more uniform meshlets reduce render times. We saw this when sorting the NVIDIA meshlet collections in Figure 10.

Inspecting Table 5 in conjunction with Table 8 reveals the correlation between high primitive fill and better performance. It is interesting to explore the interaction between average primitive fill and vertex completeness by inspecting the k-medoids and the NVIDIA meshlet collections. For the Bunny mesh, we see an example where the average primitive fill on the NVIDIA meshlet collection is so low that the high average vertex fill cannot compensate for it. This demonstrates that one should not only optimize around one heuristic but take both into account. For the Happy mesh, the NVIDIA collection performs better than k-medoids, showing that vertex completeness is more important. For the Dragon mesh, the tables have turned, and the k-medoids collection, with a better balance between the two, preforms best. This interaction tells us that it is important to prioritize both vertex completeness and primitive fill. We also observe that it is hard to have a high primitive fill without having nearly vertex-complete meshlets.

Striving for cullable meshlets is the third heuristic. Our experiments show that cullable meshlets can help balance out larger meshlet collections. Figure 11 exemplifies how meshlet collections slightly enlarged to increase cullability can indeed result in better performance. It does however not seem to affect performance as much as vertex completeness or maximizing the primitive fill.

The Skull and Nobby meshes produced some surprising results for some of the meshlet generation strategies. It is surprising that Kapoulkine's algorithm did not perform best on the Skull, as the data shows more culling and less meshlets. Perhaps the difference is that our method builds meshlets along the z-axis of the skull as opposed to from the middle and out, which could affect vertex loading, overdraw, and cache misses on the GPU.

Nobby shows that some meshes will be exceptions to the rule. It will be possible to find meshes where these heuristics and metrics break down. In fact, tuning one aspect of meshlet generation affects all the other aspects. The metrics, and indeed most of the factors we explore in this paper, are highly correlated, and this can make it hard to isolate different aspects as they affect each other. Two collections of meshlets might differ in efficiency even if almost all meshlets are packed to capacity in both collections. Because of this, it becomes even more desirable to have an algorithm that is simple to implement. The greedy algorithm proves to be quite useful in practice as it achieves good render times across the meshes while also being simple to implement.

Lastly, we conducted a small exploratory experiment that compared different ways of packing the meshlet descriptor data. Interestingly, we find that the monolithic descriptor performs quite well. This is certainly interesting. The monolithic descriptor uses a simpler buffer setup, and by using one descriptor per shader stage, it becomes possible to add more meta data if desired.

7. Conclusion

We find, quite simply, that, on the NVIDIA hardware, meshlet collections that minimize the number of meshlets and maximize the triangles in each meshlet perform best. Meshlets have vertex and primitive limits; in this paper we used the suggested 64 vertices and 126 triangles. These limits can differ between GPUs, so it is important to look up the manufacturer-suggested limits. However, our approach is not dependent on these limits but simply fills up meshlets until the limits are met. It could be an interesting extension of this work to test the algorithms on other GPU architectures. To facilitate this, we have uploaded our code to https://github.com/Senbyo/meshletmaker.

Because the triangle limit is greater than the vertex limit, we need to build the meshlets with a large emphasis on vertex reuse. Even when doing this, it is hard to not hit the vertex limit before the triangle limit. So if we want to achieve both a high vertex and triangle fill, it becomes absolutely paramount that a meshlet collection achieves a high average vertex fill. With a high vertex fill we can better take advantage of vertex reuse to also achieve a high average primitive fill. Because of this, we recommend the following strategy for optimizing meshlet generation: *make the meshlets vertex-complete first and then maximize the primitive fill.* The combination of these two will create meshlets with large vertex reuse and locality, while also minimizing the total number of meshlets that are required to represent a mesh. Finally, we of course recommend to *strive for cullable meshlets*, but not at the cost of a too big increase in meshlet collection size. We found that performance rather quickly drops when the meshlet collections grow in size. Lastly, sorting the meshlet collection to promote more uniform workloads across the GPU can also increase performance.

We also explored other properties of both the mesh shading pipeline and the meshlet collections. We found that high uniformity in the meshlet collections promotes even workload across processors on the GPU, which yields better render times. Different meshlet descriptors do not have the biggest impact on render times, so working with monolithic meshlets could prove to be a good choice for scientific visualization where rendering is done on distributed systems. As an interesting topic for future work, descriptors that require less data unpacking in the mesh shader could yield improved render performance, and since dividing descriptors into two also did not affect performance too much, it could be interesting to explore whether new useful meta data could be added.

Acknowledgements

We would like to thank Rasmus Emil Christensen and Emil Toftegaard Gæde for the initial investigation into k-medoids–based clustering of triangle meshes. This research was funded by Advokat Bent Thorbergs Fond (award no. 66.531).

References

AAGE, N., ANDREASSEN, E., LAZAROV, B. S., AND SIGMUND, O. 2017. Giga-voxel computational morphogenesis for structural design. *Nature 550*, 7674, 84–86. URL: https://doi.org/10.1038/nature23911. 14

AAGE, N., SIGMUND, O., LAZAROV, B. B., AND ANDREASSEN, E., 2020. TopWingData. DTU Data. URL: https://doi.org/10.11583/dtu.12581615.v1. 14

ARKIN, E. M., HELD, M., MITCHELL, J. S., AND SKIENA, S. S. 1996. Hamiltonian triangulations for fast rendering. *The Visual Computer 12*, 9, 429–444. URL: https://doi.org/10.1007/BF01782475. 3

BÆRENTZEN, A., AND ROTENBERG, E. 2021. Skeletonization via local separators. *ACM Transactions on Graphics 40*, 5, 187:1–187:18. URL: https://doi.org/10.1145/3459233. 11

CHOW, M. 1997. Optimized geometry compression for real-time rendering. In *Proceedings of Visualization '97*. IEEE Press, 347–354. URL: https://doi.org/10.1109/VISUAL.1997.663902. 4

CIGOLLE, Z. H., DONOW, S., EVANGELAKOS, D., MARA, M., MCGUIRE, M., AND MEYER, Q. 2014. A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques 3*, 2 (April), 1–30. URL: http://jcgt.org/published/0003/02/01/. 6

DALLY, W. J., KECKLER, S. W., AND KIRK, D. B. 2021. Evolution of the graphics processing unit (GPU). *IEEE Micro 41*, 6, 42–51. URL: https://doi.org/10.1109/MM.2021.3113475. 3

DEERING, M. F., AND NELSON, S. R. 1993. Leo: A system for cost effective 3D shaded graphics. In *SIGGRAPH '93*, ACM, 101–108. URL: https://doi.org/10.1145/166117.166130. 4

DEERING, M. 1995. Geometry compression. In *SIGGRAPH '95*, ACM, 13–20. URL: https://doi.org/10.1145/218380.218391. 4

DILLENCOURT, M. B. 1996. Finding Hamiltonian cycles in Delaunay triangulations is NP-complete. *Discrete Applied Mathematics 64*, 3, 207–217. URL: https://doi.org/10.1016/0166-218X(94)00125-W. 3

ENGLERT, M. 2020. Using mesh shaders for continuous level-of-detail terrain rendering. In *ACM SIGGRAPH 2020 Talks*, ACM, 44:1–44:2. URL: https://doi.org/10.1145/3388767.3407391. 5

EVANS, F., SKIENA, S., AND VARSHNEY, A. 1996. Optimizing triangle strips for fast rendering. In *Proceedings of Seventh Annual IEEE Conference on Visualization*, IEEE Press, 319–326. URL: https://doi.org/10.1109/VISUAL.1996.568125. 4

FORSYTH, T., 2006. Linear-speed vertex cache optimisation. Tom Forsyth's Starkly Functional Web Page, September 28. Accessed April 4, 2022. URL: https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html. 4

HAINES, E. 2006. An introductory tour of interactive rendering. *IEEE Computer Graphics and Applications 26*, 1, 76–87. URL: https://doi.org/10.1109/MCG.2006.9. 3

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH '99*, ACM/Addison-Wesley, 269–276. URL: https://doi.org/10.1145/311535.311565. 3, 4

JENSEN, M. B., JACOBSEN, E. I., FRISVAD, J. R., AND BÆRENTZEN, J. A. 2021. Tools for virtual reality visualization of highly detailed meshes. In *VisGap—The Gap between Visualization Research and Visualization Software*, The Eurographics Association, 19–26. URL: https://doi.org/10.2312/visgap.20211088. 2, 5

KAPOULKINE, A., 2017. meshoptimizer. GitHub. Accessed April 4, 2022. URL: https://github.com/zeux/meshoptimizer. 1, 2, 5, 9

KARIS, B., STUBBE, R., AND WIHLIDAL, G. 2021. A deep dive into nanite virtualized geometry. In *Advances in Real-Time Rendering in Games: Part I*, N. Tatarchuk, Ed., ACM SIGGRAPH 2021 Courses. ACM. Accessed April 4, 2022. URL: https://advances.realtimerendering.com/s2021/index.html. 5

KAUFMAN, L., AND ROUSSEEUW, P. J. 1990. *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley & Sons. URL: https://doi.org/10.1002/9780470316801. 1, 11, 13

KERBL, B., KENZEL, M., IVANCHENKO, E., SCHMALSTIEG, D., AND STEINBERGER, M. 2018. Revisiting the vertex cache: Understanding and optimizing vertex processing on the modern GPU. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 2 (August), 29:1–29:16. URL: https://doi.org/10.1145/3233302. 4

KUBISCH, C., 2018. Introduction to Turing mesh shaders. NVIDIA Developer Technical Blog, September 17. URL: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/. 2, 4

KUBISCH, C., 2018. Vulkan & OpenGL CAD mesh shader sample. GitHub. Accessed April 4, 2022. URL: https://github.com/nvpro-samples/gl_vk_meshlet_cadscene. 1, 2, 7, 8

KUBISCH, C., 2020. Using mesh shaders for professional graphics. NVIDIA Developer Technical Blog, December 8. URL: https://developer.nvidia.com/blog/using-mesh-shaders-for-professional-graphics/. 2

LEMPIAINEN, J., 2020. Meshlete: Chop 3D objects to meshlets. GitHub. Accessed April 4, 2022. URL: https://github.com/JarkkoPFC/meshlete. 5

LIN, G., AND YU, T. P.-Y. 2006. An improved vertex caching scheme for 3D mesh rendering. *IEEE Transactions on Visualization and Computer Graphics 12*, 4, 640–648. URL: https://doi.org/10.1109/TVCG.2006.59. 4

LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2, 39–55. URL: https://doi.org/10.1109/MM.2008.31. 4

NEFF, T., MUELLER, J. H., STEINBERGER, M., AND SCHMALSTIEG, D. 2022. Meshlets and how to shade them: A study on texture-space shading. *Computer Graphics Forum 41*, 2, 277–287. URL: https://doi.org/10.1111/cgf.14474. 5

REBENITSCH, L., AND OWEN, C. 2016. Review on cybersickness in applications and visual displays. *Virtual Reality 20*, 2, 101–125. URL: https://doi.org/10.1007/s10055-016-0285-9. 2

SANDER, P. V., NEHAB, D., AND BARCZAK, J. 2007. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics 26*, 3, 89:1–89:10. URL: https://doi.org/10.1145/1276377.1276489. 4, 9

SANTERRE, B., ABE, M., AND WATANABE, T. 2020. Improving GPU real-time wide terrain tessellation using the new mesh shader pipeline. In *Nicograph International (NicoInt 2020)*, IEEE Press, 86–89. URL: `https://doi.org/10.1109/NicoInt50878.2020.00025`. 5

UNTERGUGGENBERGER, J., KERBL, B., PERNSTEINER, J., AND WIMMER, M. 2021. Conservative meshlet bounds for robust culling of skinned meshes. *Computer Graphics Forum 40*, 7, 57–69. URL: `https://doi.org/10.1111/cgf.14401`. 5

WALBOURN, C., 2014. DirectXMesh geometry processing library. GitHub. Accessed April 4, 2022. URL: `https://github.com/microsoft/DirectXMesh`. 5

WIHLIDAL, G., 2016. Optimizing the graphics pipeline with compute. Game Developer Conference 2016. Accessed April 4, 2022. URL: `https://www.gdcvault.com/play/1023463/Optimizing-the-Graphics-Pipeline-With`. 5

Author Contact Information

Mark Bo Jensen
Technical University
of Denmark
Richard Petersens Plads
324, 180
Lyngby, DK-2800, Denmark
mboje@dtu.dk
https://senbyo.github.io

Jeppe Revall Frisvad
Technical University
of Denmark
Richard Petersens Plads
324, 160
Lyngby, DK-2800, Denmark
jerf@dtu.dk
https://people.compute.dtu.dk
/jerf/

J. Andreas Bærentzen
Technical University
of Denmark
Richard Petersens Plads
324, 160
Lyngby, DK-2800, Denmark
janba@dtu.dk
https://people.compute.dtu.dk
/janba/

M. B. Jensen et al., Performance Comparison of Meshlet Generation Strategies, *Journal of Computer Graphics Techniques (JCGT)*, vol. 12, no. 2, 1–1, 2023
http://jcgt.org/published/0012/02/01/

Received:	2022-07-13		
Recommended:	2022-12-13	Corresponding Editor:	Natalya Tatarchuk
Published:	2023-12-08	Editor-in-Chief:	Marc Olano