

Towards Practical Meshlet Compression

Bastian Kuth¹ Max Oberberger² Felix Kawala^{1,2} Sander Reitter^{1,2} Sebastian Michel¹ Matthäus Chajdas² Quirin Meyer¹
¹Coburg University of Applied Sciences and Arts, Germany ²AMD, Germany

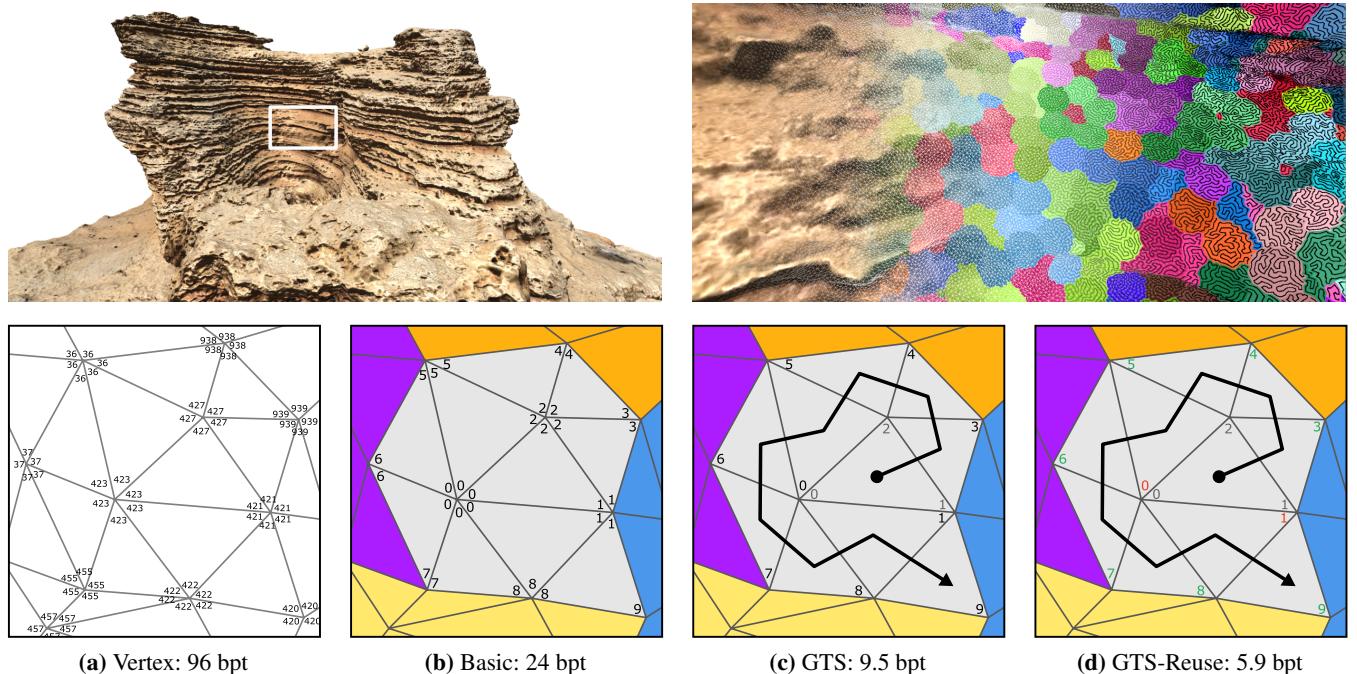


Figure 1: (a) The conventional vertex pipeline requires 3×32 bits per triangle (bpt), but, (b) meshlet-triangles only require 3×8 bpt. (c) We compress mesh triangles to 9.5 bpt by encoding meshlets into optimal generalized triangle strips (GTSSs). (d) We re-order vertices such that their first appearance needs only one bit (green) and only store reused vertex indices (red) explicitly.

Abstract

We propose a codec specifically designed for meshlet compression, optimized for rapid data-parallel GPU decompression within a mesh shader. Our compression strategy orders triangles in optimal generalized triangle strips (GTSSs), which we generate by formulating the creation as a mixed integer linear program (MILP). Our method achieves index buffer compression rates of 16:1 compared to the vertex pipeline and crack-free vertex attribute quantization based on user preference. The 15.5 million triangles of our teaser image decompress and render in 0.59 ms on an AMD Radeon RX 7900 XTX.

Keywords: geometry compression, mesh shaders, real-time rendering

1. Introduction

Mesh shaders are a recent addition to the set of programmable shaders used for rendering. They replace the vertex, tessellation, and geometry stages of the *vertex shader pipeline*. The resemblance

to a compute shader allows for greater flexibility. With meshes becoming more complex, compression methods keep their memory footprint tangible. As decompression is usually a slow and serial process, specialized techniques are required for massively parallelized hardware like graphics processing units (GPUs). In this work, we adapt and extend these techniques to the context of a mesh shader. We make the following *contributions*:

- *Meshlet topology compression schemes.* We reduce the meshlet index buffer memory footprint to ca. 5.9 bpt. We propose a slightly faster compression scheme at ca. 9.5 bpt.
- *Mesh shader decompression.* We propose real-time mesh-shader-based decompressing algorithms that outperform the conventional vertex pipeline.
- *Self-contained, crack-free vertex attribute compression.* By duplicating meshlet boundary vertices, we keep our meshlets self-contained which allows for fast, spatially-local memory access. We show a crack-free quantization for attribute compression.
- *Optimal GTS construction.* Finding the optimal GTS is NP-complete. Nonetheless, we show that it is feasible to find the optimum for common meshlet sizes using a MILP solver.

Our goal is to provide a mesh-shader-based meshlet decompression technique that is as least as fast the vertex-pipeline while significantly reducing the GPU memory footprint. Our approach has the following *limitations*: We only account for triangle rasterization and omit other mesh types and render methods. Due to our tight performance budget, we make limited use of attribute coherence in a meshlet. Finally, methods for optimally compressible meshlet generation remain future work.

2. Background and Previous Work

A vertex is an *index* V_i referencing an element in a *vertex buffer*. Vertex-buffer elements contain *vertex attributes* like positions, normals, texture coordinates, etc. An *index buffer* is a sequence of vertex indices V_i that describe the connectivity. For *triangle lists*, triplets of indices from a triangle. They have a relatively high memory requirement, but are supported by GPUs.

Triangle strips describe a path over a mesh. A path enters a triangle (V_a, V_b, V_c) through the edge (V_a, V_b) . The next triangle can either be across the *right edge* (V_b, V_c) or the *left edge* (V_c, V_a) . For *alternating triangle strips* (ATSs), the path always alternates between right and left edge. Then, three consecutive indices in an index buffer form a triangle. Except for the first one, each triangle requires only one new index. *Generalized triangle strips* (GTSs) are more flexible: for each triangle, one so-called *L/R-flag* per triangle discriminates whether the path continues across the left or right edge [VdFG99]. Fig. 1 shows an example of a GTS.

While ATSs result in long triangle bands, GTSs avoid them and allow layouts that better fit GPU run-time behavior [Kil08, Sec. 7.2]. Further, typically GTSs require fewer strips than ATSs and have a smaller memory footprint. Since GPUs do not support GTSs directly, we build upon the algorithm proposed by Meyer et al. [MKSS12] to decode them into triangle lists.

Geometry compression handles the massive and increasing amount of data in real-time computer graphics. Several overview reports cover this vast field [AG03, PKJ05, MLDH15]. While many methods target offline decompression, we narrow our scope to mesh representations suitable for GPU rendering.

Calver [Cal02] describes vertex shader attribute de-quantization using 8- and 16-bit integers for each attribute channel. Purnomo et al. [PBCK05] assign a fixed bit-budget for all vertex attributes. Every attribute channel is quantized with an arbitrary number of bits

using a pre-process that compares the rendering error for different bit-allocations. Kwan et al. [KXW^{*}18] store vertex attributes as 2D block-compressed textures.

For positions, quantization levels are typically determined empirically [Dee95], usually at 8 – 12 bits per component [PKJ05, AG03]. To compactly represent positions, Lee et al. [LCL10] first align a mesh on a global grid to prevent cracks. Next, they decompose a mesh until each sub-mesh does not require more than 8 bits along each spatial axis. Meyer et al. [MSG11] dynamically add and remove bits view-dependently.

For normals, various compression methods exist: Deering [Dee95] proposed spherical parameterization methods, which, however, use expensive trigonometric functions. Meyer [Mey12] carefully analyzes the error of various parameterization schemes. Octahedron unit vectors turn out to be an effective method for vertex-shader decompression [MSS^{*}10]. Cigolle and colleagues [CDE^{*}14] provide effective implementations. Keinert et al. [KISS15] propose a fast and precise unit vector decomposition method based on a Fibonacci mapping.

Frey and Herzeg [FH11] convert three perpendicular tangent-space unit-vectors used as vertex attributes required for normal mapping to a quaternion. They decompress the corresponding tangent-space in a vertex shader. Recently, methods for blend-attributes compression were presented [KM21, PKM22].

In his pioneering work, Deering [Dee95] introduced real-time index buffer compression. Offline methods like Edgebreaker [Ros99] or the Cut-Border Machine [GS98] compress close to the minimum of ca. 1.62 bpt [Tut62]. Jakob et al. [JBG17] provide a fast, but non-real-time, GPU-implementation of the Cut-Border Machine. Meyer et al. [MKSS12] describe a real-time algorithm to quickly decompress a GTS using data-parallel scans. Karis et al. [KSW21] replace these scans by faster bit-scans, but overall, do not consider their approach to be fast enough for frame-by-frame decompression.

Schäfer et al. [SPM^{*}12] simplify a mesh with edge-collapses. The lost geometric information is re-sampled and compactly stored. During run-time it is reconstructed using hardware-tessellation patterns. Maggiordomo and coworkers [MMT23] use a lossy algorithm to convert a highly detailed mesh into the recently introduced micro-mesh [Nvi22] data structure. Both approaches significantly reduce memory and rendering time, but change topology, which might not be acceptable for all applications.

Laced Ring [GLLR11] is a compact data structure for triangle mesh adjacency information. It targets mesh processing algorithms that require finding triangle neighbors or fans around a vertex. Hence, it bears some inherent overhead when only used for mesh compression. However, Mlakar et al. [MSS24] showed that is also suited for meshlet decompression and can achieve real-time rates.

3. Mesh-Shader

Amplification- and mesh-shaders provide a compute-shader-based programming model to hand triangles to the raster stage. Previously, the *vertex pipeline* handled this task. From the central processing unit (CPU), the programmer launches multiple mesh-shader *thread groups* of up to 128 threads. Hardware internally

schedules the thread groups into *waves* of 32 or 64 threads, which then run on a *compute unit*. Each shader thread-group outputs a small mesh – called *meshlet* – to the raster stage. A meshlet consists of a triangle list stored in a local vertex-buffer with per-vertex attributes and an index-buffer whose elements point into the local vertex-buffer. Each meshlet has a limit \tilde{V} vertices and \tilde{T} triangles, currently set to 256 in DirectX [Mic23]. An *amplification shader* stage may run before the mesh-shader stage. It has the ability to dispatch or suspend mesh-shader thread groups. We use the amplification shader for a *cone culling* optimization: the triangle normals of a meshlet form a cone that we store using an axis and an opening angle. Then, the amplification shader can quickly assess, if all triangles are back-facing and then cull the entire meshlet.

Each mesh-shader work-group needs to read in a meshlet from GPU memory. To obtain meshlets in a pre-process, a meshlet split algorithm tries to find coherent groups of triangles such that the total number of vertices V and triangle T per such group, is smaller than or equal to the given maximum \tilde{V} and \tilde{T} . We observe that a typical meshlet is a connected patch of 2-manifold triangles, where $V < T < 2V$ holds. Thus, for configurations where a meshlet split algorithm is given limits of $\tilde{T} = \tilde{V}$, the limiting factor for the meshlet size is \tilde{T} , which is reached for most meshlets. Vice versa, for configurations $\tilde{T} = 2\tilde{V}$, the limiting factor for the meshlet size is \tilde{V} , which is reached for almost all meshlets, while the limit \tilde{T} is rarely reached. The workload per vertex, e.g., attribute transformations, is usually greater than the workload per triangle, e.g., loading three indices. Therefore, to achieve higher GPU utilization, it makes sense to set the limits such that V is likely to reach a \tilde{V} that is a multiple of the hardware’s wave size.

As a meshlet index only references up to 256 unique vertices, the basic mesh-shading pipeline stores 8 bit per index, instead of 32 bits of a global index buffer of a vertex pipeline. To achieve this, a meshlet stores a mesh-global offset to where its vertex attributes start in the buffer. As a consequence, all the vertex attributes lying on the border between meshlets have to be duplicated, as each meshlet needs it in its local attribute space. To reduce this data redundancy, meshlet builders can try to minimize the average border length of all meshlets. It is possible to avoid attribute duplication entirely by using a secondary index buffer that references the attributes shared over multiple meshlets [Kub18]. While this would make the overall compression smaller, it makes the implementation more complex. It also disrupts memory-locality of attributes, and thus makes memory access less coherent. Further, it requires extra indirect memory accesses and makes streaming of individual meshlets more complex. Finally, we want to utilize the geometric coherence between attributes of a meshlet to better quantize attributes in Sec. 4.4. Therefore, we prefer *self-contained meshlets*, holding all attributes tightly in memory. For our test cases, we found $\tilde{V} = 128$ and $\tilde{T} = 256$ to be optimal. With $\tilde{V} = 128$, 7 bit indices would suffice, but we choose 8-bit for faster byte-aligned memory access.

4. Compression

In this section, we formulate a MILP for finding the optimal GTS. We then describe the strip encoding and parallel decoding. We close this section with our crack-free attribute quantization.

4.1. Optimal Generalized Triangle Strip

To compress a meshlet’s index buffer, we represent triangles as a GTS. For the best compression ratio, we need to find strips over the whole meshlet with the minimum possible number of restarts. This problem is, however, NP-complete [AHMS96] and many approximations exist, but see Vaněček’s and Kolingerová’s overview [VK07].

Many practical optimization problems can be modeled as

$$\max \sum_i c_i v_i \text{ such that } \mathbf{A}\vec{v} \preceq \vec{b}, v_i \geq 0, \quad (1)$$

with $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\vec{b} \in \mathbb{R}^m$, $\vec{c} \in \mathbb{R}^n$, n the number of variables to be optimized, m the number of constraints, and \preceq the component-wise \leq operator. If $v_i \in \mathbb{R}$, Eq. 1 is a linear program (LP), if $v_i \in \mathbb{Z}$ an integer linear program (ILP), and a mixed integer linear program (MILP) when there is both. Estkowski et al. [EMX02] formulate ATS computation as an ILP. We extend their approach to work for GTSs. Consider the *dual graph* of the triangle mesh. A *dual graph edge* connects two triangles, thus the *dual graph nodes*, that share an edge. Let $\vec{v} = [\vec{x}, \vec{y}]^\top$ in Eq. 1. For each such edge i , the variable $x_i \in \{0; 1\}$ defines, whether it is part of the strip (1) or not (0). The x_i are the relevant solutions to our MILP. To minimize the required number of strip restarts, we need to maximize the number of edges that are part of the strip: $\max \sum_i x_i$. As this condition alone does not create valid triangle strips, we add two types of constraints:

No-fork Constraint Since a manifold triangle has at most three neighboring triangles (one per edge) each dual-graph node has at most three dual-graph edges x_b, x_l, x_r . If $x_b + x_l + x_r = 3$, all dual-graph edges would belong to the strip. In such a case the strip would fork at this triangle, which is not possible. Therefore, similar to Estkowski et al. [EMX02], we constrain

$$x_b + x_l + x_r \leq 2. \quad (2)$$

Anti-cycle Constraint To avoid unwanted cycles, Estkowski et al. [EMX02] add one constraint per potential cyclic ATS. For ATSSs, this is tangible because enumerating all cycles is $\mathcal{O}(n)$, but infeasible for GTSs. Instead, solvers like Gurobi [Gur23] use *lazy constraints*, which can be dynamically added once a potential solution is found. We observed that lazy anti-cycle constraints do not yield a solution within a reasonable time- or memory-frame. As an alternative approach, Cohen [Coh19] derived anti-cycle constraints for common graph problems. For finding a non-cyclic GTS, we modify and simplify them to the following: to each edge connecting some nodes A and B , we assign two variables $y_{\bar{ab}}$ and $y_{\bar{ba}}$ on either side of the edge, where $y_i \in \mathbb{R}_{\geq 0}$. We constrain the sum of the two edge variables to an arbitrary constant value $F \in \mathbb{R}_{>0}$, e.g., $F = 1$:

$$y_{\bar{ab}} + y_{\bar{ba}} = F \cdot x_{\bar{ab}}. \quad (3)$$

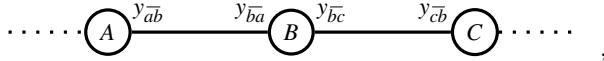
Thus, if the edge is part of the strip, so when $x_{\bar{ab}} = 1$, the sum $y_{\bar{ab}} + y_{\bar{ba}}$ must be F . Otherwise, if the edge is not part of the strip, both variables are zero. Additionally, at each node, we constrain the sum of the adjacent edge flow variables, e.g., for a node A ,

$$y_{\bar{ab}} + y_{\bar{ac}} + y_{\bar{ad}} < F. \quad (4)$$

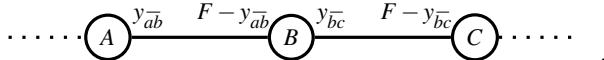
As strict inequality are not allowed in LPs programs, we write

$$y_{\bar{ab}} + y_{\bar{ac}} + y_{\bar{ad}} \leq F - \epsilon$$

instead. Consider a strip section:



which can be rewritten according to Eq. 3:



With Eq. 4, it is given that $y_{ab} > y_{bc}$, or the other way round $y_{ba} < y_{cb}$. As these inequalities cannot hold for a cyclic strip, we successfully prevent cycles with these additional constraints.

Finally, we feed our MILP to the Gurobi [Gur23] and SCIP [BBC*21] solvers to find an optimal solution, as shown in our supplemental code.

4.2. Strip Encoding

As established in Sec. 2, an L/R-flag per triangle marks the edge reused from the previous triangle. As we decompress all triangles in parallel, a thread cannot access the decompressed previous triangle. Instead, each thread must find the correct indices from earlier in the strip. See Fig. 2 for reference: the longer a GTS spins around a vertex in a fan-like manner, the further back the required index lies: here, triangle 8 requires the index of triangle 4, which is 0.

Meyer et al. [MKSS12] use a max-scan to find this offset. Initial experiments with wave-intrinsics slowed our decompression. Thus, we use GPUs-bit-level instructions on the L/R flags-for this search, similar to Nanite [KSW21]: With *firstbithigh* [Mic20], we find the index of the first set bit of a word, starting at the most significant bit. To make use of it, we shift and combine the flag words in a way that the most significant bit is the flag of the current triangle. The next bit contains the flag of the triangle before the current triangle and so on. In case the flag of the current triangle is set, we flip all bits before calling *firstbithigh*, to search for the first unset bit. For the rare case that the local fan exceeds 31 triangles, thus 32 bits of a word are not enough, we iteratively check previous words of the bit-flags. To enable hardware triangle-back-face culling, we flip the triangle orientation depending on the L/R-flag.

Even our optimal GTS algorithm may need more than one strip per meshlet. Then, we require a strip *restart*. Using an individual restart bit per triangle would make the implementation more complex, and, as restarts are rare, they increase the memory footprint. Similar to Meyer et al. [MKSS12], we use four degenerate triangles to emulate a restart. In our evaluation, we show that degenerate triangles do not contribute much to the total run-time cost. This increases the number of triangles in the strip. In rare cases, we exceed the $\tilde{T} = 256$ limit and we must split the meshlet.

4.3. Index Reuse Packing

As the vertex attributes referenced by the index buffer are local per meshlet, we reorder them such that new vertices appear in the strip in ascending order. This means a strip always starts with $(0, 1, 2, \dots)$. Thus, the first triangle of a meshlet does not have to be stored explicitly. Furthermore, the majority of indices are just increments of the previous index. Meyer et al. [MKSS12] make

use of this redundancy by storing one bit per triangle that indicates whether the index appears before in the strip. Indices which are increments of the previous index are marked with a 1 increment flag. Indices which are not increments, thus already appeared in the strip, are marked as reuse with a 0 increment flag. A prefix add scan over all flags, which is a common per-wave intrinsic, allows us to retrieved all incrementing indices. If the flag of the current triangle t is 1, the result of the scan s is the current index. If the flag is 0, an additional reuse array is accessed at location $t + 1 - s$, see Fig. 2. To speed up computation, we again use a bit instruction over the increment flags and avoid synchronization between threads. Thus, we count the number of set bits in a word, called *countbits* [Mic20].

4.4. Crack-Free Fixed-Point Vertex Attribute Quantization

While developing our compression scheme, we observed that mesh shaders have a limited compute budget. There, we have little room for sophisticated attribute compression. As our goal is to beat the vertex pipeline, we settle with quantization. We must, however, make sure that duplicate vertices along a meshlet boundary map to the same values. Otherwise, we would get unwanted cracks.

The attributes of a vertex are organized as an *attribute vector* $\vec{A} \in \mathbb{R}^n$. A component A_i is called *attribute channel*. For example, if a vertex consists of a position $\in \mathbb{R}^3$, a normal $\in \mathbb{R}^3$, and a texture coordinate $\in \mathbb{R}^2$, $\vec{A} \in \mathbb{R}^8$. For convenient vertex-processing, attributes are usually floating-point quantized. But the logarithmic quantization-curve of floating-point numbers does not concur with typical distributions found for attributes. Therefore, it is memory-wasteful. Instead, we use a simple uniform quantization with b bits for each attribute channel. We find $b = 16$ bits for each attribute channel a sensible choice, because it allows fast memory-aligned attribute fetches and we found no visual deviations for our test meshes. Further, the precision is higher than the 16-bit floating-point format commonly used in practice.

To avoid cracks, we propose to first map the meshlets to a global anisotropic grid with the following uniform spacings along each axis: For each attribute channel i , such as the x coordinate of the position, we find the meshlet with the largest extent w_i . The sample spacing of the global grid for channel i is then $\Delta_i = w_i / (2^b - 1)$. In general, the global grid requires more than b bits for channel and is therefore more precise than the local grid. For each meshlet, we store its lowest value L_i for each channel i with respect to the global grid. The values of the attribute channel are stored relative to L_i . This guarantees that b bits are sufficient for each attribute channel. For decompression, an integer addition reconstitutes the attribute values to the global grid. Then we map the values back to their original floating-point representation, used for standard vertex-processing. This scheme is simple and suits the tight compute budget of mesh-shaders.

Assuming independent and identically distributed attributes, we get a greater or equal information content for the attribute channel i than b . Let the global mesh extent of the attribute channel i be W_i . The information content of this channel is then $\log_2(W_i / \Delta_i) \geq b$ bits for this channel, but see Tab. 3 for concrete values.

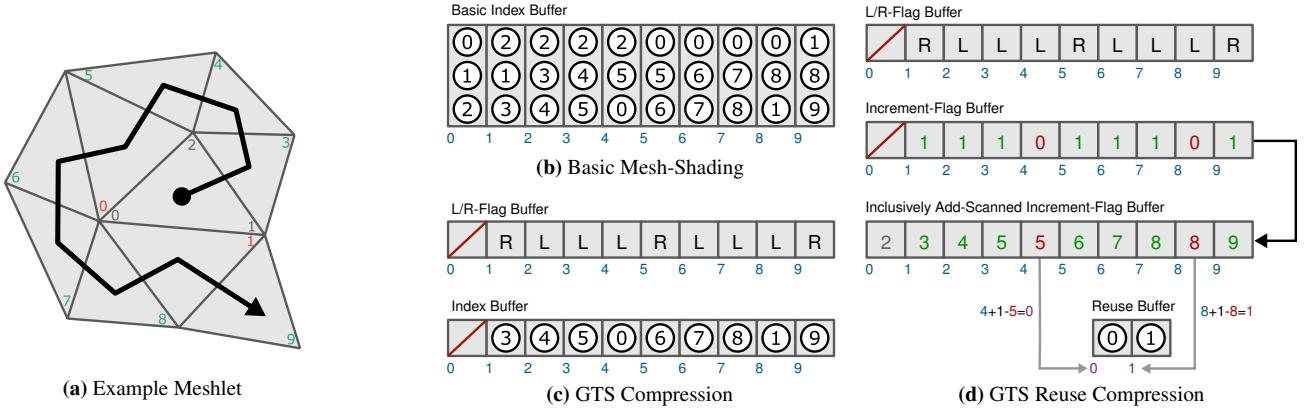


Figure 2: Strip Encoding. **(a)** Example meshlet with vertex indices at the corners. The polyline denotes the triangle strip. **(b)** For *Basic Mesh-Shading*, triangles use three indices stored in the *Basic Index Buffer*. **(c)** Our meshlet *GTS Compression* consists of one index per triangle in an *Index Buffer*. The indices 0, 1, 2 are not explicitly stored. The *L/R-Flag Buffer* stores a flag denoting the direction of the strip. **(d)** Our meshlet *GTS-Reuse Compression* makes use of incrementing vertex indices (green) stored in an *Increment-Flag Buffer*. In addition to the *L/R-Flag Buffer*, the reused vertex indices (red) have to be stored in the *Reuse Buffer*. We compute *Inclusively Add-Scanned Increment-Flag Buffer* during decompression. In case an index is reused, we compute the index location in the *Reuse Buffer* with additions and subtractions.

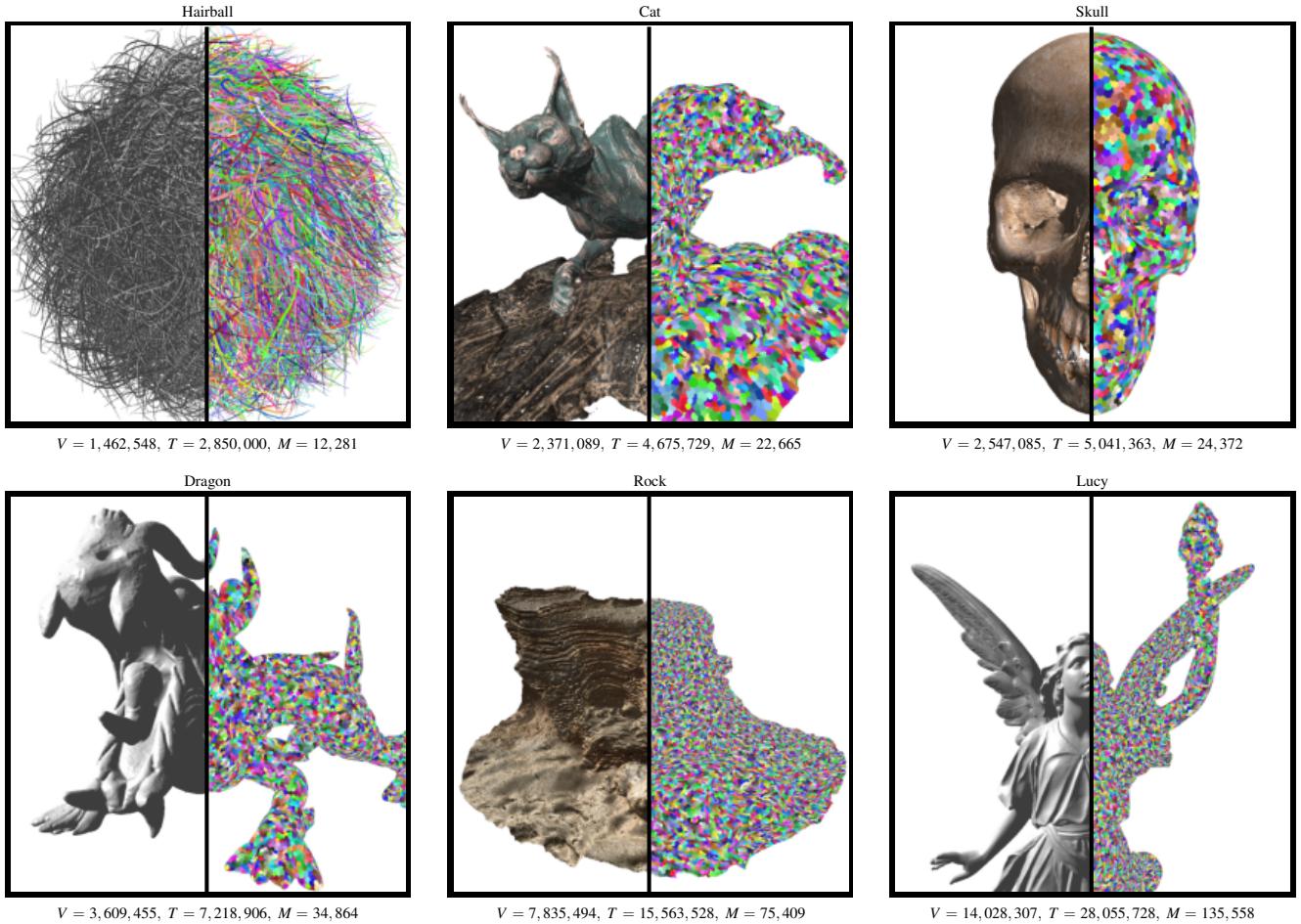


Figure 3: Test Meshes. V denotes the number of vertices of the input mesh, T the number of triangles and M the number of meshlets coming from Meshoptimizer. Meshlets are visualized with randomized colors.

	Gurobi	SCIP	ETA
computation time	915.46 s	8,122.56 s	10.52 s
GTS render time	0.58 ms	0.58 ms	0.59 ms
GTS restarts	123,784	123,784	185,480
degenerate triangles	495,136	495,136	741,920
additional meshlets	3 / 75,412	3 / 75,412	4 / 75,413

Table 1: GTS comparison for the *Rock* mesh. We compare the optimal Gurobi and SCIP solutions against the sub-optimal ETA. The CPU computation uses one thread per meshlet and was measured on an AMD Ryzen 9 7950X (16C/32T).

5. Results and Discussion

We evaluate the performance of our methods on the meshes of Fig. 3. Except for *Hairball* and *Cat*, our meshes come from scanning procedures. *Hairball* demonstrates that the graphics pipeline can be bound by other factors such as overdraw. Meshlets were generated using the Meshoptimizer library [Kap23] with $\tilde{V} = 128$ and $\tilde{T} = 256$. For the vertex pipeline experiments, we use vertex-cache and vertex fetch-optimization from Meshoptimizer.

Tab. 1 compares our optimal GTSs achieved with our MILP of Sec. 4.1 with Gurobi and SCIP against the sub-optimal strip of our Enhanced Tunneling Algorithm (ETA) [PS06] implementation. As expected, the ETA is orders of magnitude faster, but yields results worse than the globally optimal solution provided by the MILP solvers. Although ETA requires about 1.5 times as many strip restarts, and thus degenerate triangles, the difference in render time is negligible. This also confirms the assumption that degenerate triangles do not contribute much to the run-time cost.

Tab. 2 compares the memory requirements for different compression scenarios. For the *Index Buffer*, the basic mesh shader already achieves a compression ratio of $\sim 4:1$ over the vertex pipeline. With our GTS compression of Sec. 4.2, we achieve $\sim 10:1$. By further packing the triangles with GTS-Reuse of Sec. 4.3, we achieve $\sim 16:1$. With mesh-shaders, we store *Meshlet Meta Buffer* such as buffer offsets (12 B) and cull information (16 B). Due to degenerate triangles encoding restarts, both GTS variants require extra meshlets in rare cases, where the triangle count exceeds \tilde{T} .

For *Vertex Buffer*, we use eight attributes. To leverage the advantages of self-contained meshlets described in Sec. 3, we duplicate $\sim 20\%$ of the vertices. As expected, *Fixed-point* quantization reduces the memory consumption by a factor of $\sim 2\%$ over *Floating-point* quantization. Tab. 3 shows that our crack-free global-quantization increases the precision for the positions. On the other hand, no precision is added to normal vectors, as a single meshlet with normal vectors cluttered to all directions is enough to mitigate the benefit.

Fig. 4 shows our rendering performance measurements on different GPUs. To emphasize the performance impact of the geometry stage, our Direct3D12 implementation renders to a 500 by 500 pixel framebuffer with texture mapping and Phong Shading. For more complex shading, the performance impact of the geometry stage on the total render-time decreases. As expected, the render-time grows linearly with the mesh size.



Figure 4: We compare the performance of different versions of our compression with (a) and without (b) cone-culling against the vertex pipeline (lower is better). (c) To normalize for different mesh sizes, we compare the triangle throughput per second of our compression against the vertex pipeline (higher is better).

Mesh	Index Buffer				Meshlet Buffer		Vertex Buffer			
	Vertex Pipeline	Basic Mesh Shading	GTS	GTS Reuse	Basic Mesh Shading	GTS & GTS Reuse	Vertex Pipeline	Basic Mesh Shading	GTS & GTS Reuse	Fixed
Hairball	32.6 (96)	8.15 (24)	3.21 (9.4)	2.11 (6.2)	0.328	0.336	44.6	48.0	48.1	25.3
Cat	53.5 (96)	13.4 (24)	5.33 (9.6)	3.32 (5.9)	0.605	0.605	72.4	88.5	88.5	46.4
Skull	57.7 (96)	14.4 (24)	5.71 (9.5)	3.55 (5.9)	0.651	0.651	77.7	95.2	95.2	49.9
Dragon	82.6 (96)	20.7 (24)	8.17 (9.5)	5.07 (5.9)	0.931	0.931	110	136	136	71.4
Rock	178 (96)	44.5 (24)	17.7 (9.5)	11.0 (5.9)	2.01	2.01	239	295	295	154
Lucy	321 (96)	80.3 (24)	31.8 (9.5)	19.7 (5.9)	3.62	3.62	428	529	529	278

Table 2: Memory Requirements. We compare the sizes in MiB of the *Index*, *Meshlet*, and *Vertex* buffers for the conventional *Vertex Pipeline* and our mesh-shading pipelines (*Basic Mesh Shading*, *GTS*, *GTS-Reuse*) for the respective *Mesh*. The values in parentheses are bpt. Column *Vertex Buffer* lists the memory consumption when using 8×32 bit Floating-point values per-vertex, except for column *Fixed*, where attributes have 8×16 bit fixed-point values. *Fixed* also includes the per-meshlet constants required for dequantization.

Positions			Normals			Texture Coords.	
X	Y	Z	X	Y	Z	U	V
17.6	17.6	17.1	16.0	16.0	16.0	16.1	16.0

Table 3: Information content of our crack-free global-quantization on the *Rock* mesh. We test each attribute channel of our meshlets with a fixed number of bits b , here $b = 16$. This allows for fast memory-aligned access. Mapping the resulting sample-spacings from the meshlet-local grid to the global grid results in the shown global information content for the attribute values.

To normalize different mesh sizes, Fig. 4c compares the triangle throughput per second. Our basic mesh-shading pipeline outperforms the vertex pipeline. When adding decompression, rendering is still faster than the vertex pipeline. As expected, this is not always the case for the hairball, which is bound by pixel overdraw.

Fig. 4b is the same measurement as Fig. 4a, but with cone culling enabled. At most every second meshlet faces away from the camera, but in practise, this number is lower. Therefore, we observe that cone culling improves performance by only 1.6 times at most. In the extreme case of our rock mesh, almost all meshlets face forward and cone-culling computations are in vain.

With our index and vertex buffer compression, the rendering performance is faster than the original vertex pipeline, when the pipeline is not bound by overdraw, but the memory footprint is significantly smaller. More compression would only degrade performance, which counteracts our goal of beating the vertex pipeline in speed and size. We observe that the compute capabilities of a mesh-shader are limited.

We compare against the meshlet compression scheme "Laced Wires (LW)" [MSS24]. For connectivity, LW requires 16 bpt. With ca. 5.9 bpt, our method is almost three times smaller. LW only reports numbers for positions, while we support an arbitrary number of attributes. Therefore, we average index-, position-, and meshlet-buffer sizes over all models and obtain 38 bpt for our method. LW compresses on their test corpus at an average of 37.5 bpt. Note that LW quantizes positions to an average of $3 \times 15 = 45$ bits per vertex (bpv), whereas our approach is more precise with $3 \times 16 = 48$ bpv.

When configuring our quantization algorithm to achieve LW's precision of 45 bpv for the Rock mesh, we need 13.4, 13.4, and 13.9 bits for the three position components, totaling 40.7 bpv. Due to better memory-alignment, we prefer power-of-two quantization. LW encoding timings range from "tens of minutes to hours for large scenes." We encode the Rock mesh in 24 s (ETA) and 15 minutes (Gurobi) including meshlet generation and quantization. On an Nvidia RTX 4090, LW decompresses and renders at 13.3 Giga triangles per second (Gtps) *with frustum and cone culling* optimization, only three attributes, and no textures. Our slowest approach *without meshlet culling*, with eight attributes, and textures achieves 22.7 – 26.7 Gtps and up to 36.8 Gtps with cone culling. To conclude the comparision, without culling, our approach is twice as fast, with cone-culling three times faster, it is more precise, and achieves comparable compression ratios.

6. Conclusion and Future Work

We proposed meshlet codecs. To compress the topology of a meshlet, we find the optimal GTS. Index coherence within the strip is then used for further compression. Our method compresses the input index buffer of the conventional vertex pipeline at a ratio of up to 16:1. Furthermore, we demonstrated how to perform memory-aligned and crack-free attribute quantization, while making use of the limited local value range within a meshlet. Our evaluation shows that our decompression runs faster than the vertex pipeline. As future work we see specialized meshlet builders and more compact attribute representations. We include code for GPU decoding, quantization, and our MILP in the supplemental material.

Acknowledgments

We thank Dominik Baumeister. Meshes are courtesy of Aixtrior (Rock, Skull), Morgan McGuire (Hairball), the Stanford 3D Scanning Repository (Dragon, Lucy), and Ekaterina Kozemerchak (Cat).

References

- [AG03] ALLIEZ P., GOTSMAN C.: Recent advances in compression of 3D meshes. In *Advances in Multiresolution for Geometric Modelling* (2003), Springer Berlin Heidelberg, 2.

- [AHMS96] ARKIN E. M., HELD M., MITCHELL J. S. B., SKIENA S. S.: Hamiltonian triangulations for fast rendering. *The Visual Computer* 12, 9 (1996), 429–444. 3
- [BBC*21] BESTUZHEVA K., BESANÇON M., CHEN W.-K., CHMIELA A., DONKIEWICZ T., VAN DOORNMALEN J., EIFLER L., GAUL O., GAMRATH G., GLEIXNER A., GOTTFALD L., GRACZYK C., HALBIG K., HOEN A., HOJNY C., VAN DER HULST R., KOCH T., LÜBBECKE M., MAHER S. J., MATTER F., MÜHMER E., MÜLLER B., PFETSCH M. E., REHFELDT D., SCHLEIN S., SCHLÖSSER F., SERRANO F., SHINANO Y., SOFRANAC B., TURNER M., VIGERSKE S., WEGSCHEIDER F., WELLNER P., WENINGER D., WITZIG J.: *The SCIP Optimization Suite 8.0*. Technical report, Optimization Online, December 2021. URL: http://www.optimization-online.org/DB_HTML/2021/12/8728.html. 4
- [Cal02] CALVER D.: *Vertex and Pixel Shader Tips and Tricks*. Wordware Publishing, 2002, ch. Vertex Decompression in a Shader, pp. 172 – 187. 2
- [CDE*14] CIGOLLE Z. H., DONOW S., EVANGELAKOS D., MARA M., MCGUIRE M., MEYER Q.: A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques (JCGT)* 3, 2 (April 2014). 2
- [Coh19] COHEN N.: Several Graph problems and their Linear Program formulations. working paper or preprint, Jan. 2019. URL: https://hal.inria.fr/inria-00504914_3
- [Dee95] DEERING M.: Geometry compression. SIGGRAPH '95. 2
- [EMX02] ESTKOWSKI R., MITCHELL J. S. B., XIANG X.: Optimal decomposition of polygonal models into triangle strips. In *Proceedings of the Eighteenth Annual Symposium on Computational Geometry* (New York, NY, USA, 2002), SCG '02, Association for Computing Machinery, p. 254–263. 3
- [FH11] FREY I. Z., HERZEG I.: Spherical skinning with dual quaternions and qtangents. In *ACM SIGGRAPH 2011 Talks* (2011), SIGGRAPH '11, Association for Computing Machinery. 2
- [GLLR11] GURUNG T., LUFFEL M., LINDSTROM P., ROSSIGNAC J.: Lr: compact connectivity representation for triangle meshes. *ACM Trans. Graph.* 30, 4 (2011). 2
- [GS98] GUMHOLD S., STRASSER W.: Real time compression of triangle mesh connectivity. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (1998), Association for Computing Machinery. 2
- [Gur23] GUROBI OPTIMIZATION, LLC: Gurobi Optimizer Reference Manual, 2023. URL: <https://www.gurobi.com>. 3, 4
- [JBG17] JAKOB J., BUCHENAU C., GUTHE M.: A parallel approach to compression and decompression of triangle meshes using the GPU. *Comput. Graph. Forum* 36, 5 (Aug. 2017). 2
- [Kap23] KAPOULKINE A.: meshoptimizer, 2023. URL: <https://github.com/zeux/meshoptimizer>. 6
- [Kil08] KILGARD M.: *Modern OpenGL usage: Using vertex buffer objects well*. Tech. rep., NVIDIA Corporation, 2008. 2
- [KISS15] KEINERT B., INNMANN M., SÄNGER M., STAMMINGER M.: Spherical fibonacci mapping. *ACM Trans. Graph.* 34, 6 (Oct. 2015). 2
- [KM21] KUTH B., MEYER Q.: Vertex-blend attribute compression. In *High-Performance Graphics - Symposium Papers* (2021), Binder N., Ritschel T., (Eds.), The Eurographics Association. 2
- [KSW21] KARIS B., STUBBE R., WIHLIDAL G.: A deep dive into nanite virtualized geometry. In *ACM SIGGRAPH* (2021). 2, 4
- [Kub18] KUBISCH C.: *Introduction to Turing Mesh Shaders*, 09 2018. URL: <https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/>. 3
- [KXW*18] KWAN K. C., XU X., WAN L., WONG T., PANG W.: Packing vertex data into hardware-decompressible textures. *IEEE Transactions on Visualization and Computer Graphics* 24, 5 (2018). 2
- [LCL10] LEE J., CHOE S., LEE S.: Compression of 3D Mesh Geometry and Vertex Attributes for Mobile Graphics. *JCSE* 4 (09 2010). 2
- [Mey12] MEYER Q.: *Real-Time Geometry Decompression on Graphics Hardware*. PhD thesis, 08 2012. 2
- [Mic20] MICROSOFT: *High-level shader language (HLSL)*, 08 2020. URL: <https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl>. 4
- [Mic23] MICROSOFT: *DirectX-Specs*, 2023. URL: <https://microsoft.github.io/DirectX-Specs/>. 3
- [MKSS12] MEYER Q., KEINERT B., SUSSNER G., STAMMINGER M.: Data-parallel decompression of triangle mesh topology. *Computer Graphics Forum* 31, 8 (2012), 2541–2553. 2, 4
- [MLDH15] MAGLO A., LAVOUÉ G., DUPONT F., HUDELOT C.: 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Comput. Surv.* 47, 3 (2015). 2
- [MMT23] MAGGIORDOMO A., MORETON H., TARINI M.: Micro-mesh construction. *ACM Trans. Graph.* 42, 4 (2023). 2
- [MSG511] MEYER Q., SUSSNER G., GREINER G., STAMMINGER M.: Adaptive level-of-precision for GPU-rendering. In *Vision, Modeling, and Visualization (2011)* (2011), The Eurographics Association. 2
- [MSS*10] MEYER Q., SÜSSMUTH J., SUSSNER G., STAMMINGER M., GREINER G.: On floating-point normal vectors. In *Proceedings of the 21st Eurographics Conference on Rendering* (2010), EGSR'10, Eurographics Association. 2
- [MSS24] MLAKAR D., STEINBERGER M., SCHMALSTIEG D.: End-to-end compressed meshlet rendering. *Computer Graphics Forum* 43, 1 (2024). 2, 7
- [Nvi22] NVIDIA: Displacement-MicroMap-Toolkit, 2022. URL: <https://github.com/NVIDIAGameWorks/Displacement-MicroMap-Toolkit>. 2
- [PBCK05] PURNOMO B., BIODEAU J., COHEN J. D., KUMAR S.: Hardware-compatible vertex compression using quantization and simplification. HWWS '05, Association for Computing Machinery. 2
- [PKJ05] PENG J., KIM C.-S., JAY KUO C.-C.: Technologies for 3D mesh compression: A survey. *Journal of Visual Communication and Image Representation* 16, 6 (2005). 2
- [PKM22] PETERS C., KUTH B., MEYER Q.: Permutation coding for vertex-blend attribute compression. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 1 (may 2022). 2
- [PS06] PORCU M. B., SCATENI R.: Partitioning Meshes into Strips using the Enhanced Tunnelling Algorithm (ETA). In *VRIPHYS* (2006), pp. 61–70. 6
- [Ros99] ROSSIGNAC J.: Edgebreaker: Connectivity compression for triangle meshes. *IEEE transactions on visualization and computer graphics* 5, 1 (1999), 47–61. 2
- [SPM*12] SCHÄFER H., PRUS M., MEYER Q., SÜSSMUTH J., STAMMINGER M.: Multiresolution attributes for tessellated meshes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2012), Association for Computing Machinery. 2
- [Tut62] TUTTE W. T.: A census of planar triangulations. *Canadian Journal of Mathematics* 14 (1962), 21–38. 2
- [VdFG99] VELHO L., DE FIGUEIREDO L. H., GOMES J.: Hierarchical generalized triangle strips. *The Visual Computer* 15, 1 (1999), 21–35. 2
- [VK07] VANĚČEK P., KOLINGEROVÁ I.: Comparison of triangle strips algorithms. *Computers and Graphics* 31, 1 (2007), 100–118. 3