DGF: A Dense, Hardware-Friendly Geometry Format for Lossily Compressing Meshlets with Arbitrary Topologies

JOSHUA BARCZAK, Advanced Micro Devices, Inc., USA
CARSTEN BENTHIN, Advanced Micro Devices, Inc., Germany
DAVID MCALLISTER, Advanced Micro Devices, Inc., USA

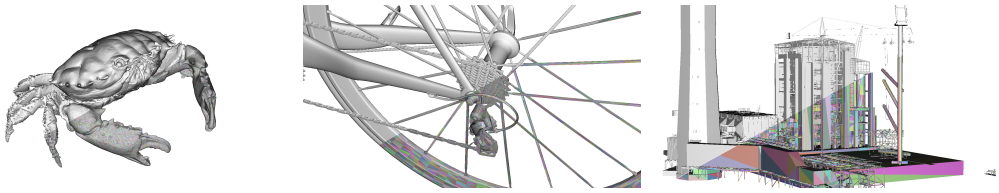


Fig. 1. A variety of models compressed using DGF. Left:Reef Crab (2.1M triangles 2.9B/triangle). Center: Bicycle with individually modeled chain links (1.7M triangles, 3.73B/triangle). Right: Powerplant with many long, axis-aligned triangles (12.7M triangles, 3.05B/triangle)

The widespread availability of hardware accelerated ray tracing solutions is driving a gradual sea-change in the real-time graphics space. The major graphics APIs now offer standardized support for accelerated ray tracing. In the same time frame, rasterization-based systems such as Nanite [Karis et al. 2021] have significantly raised geometric complexity in games. The state of the art in raster graphics now enables lossy compressed geometry representations that are decoded on the-fly during rendering. This trend conflicts with current ray tracing interfaces, which require opaque acceleration structures to be built from uncompressed input data. This paper seeks to close the gap by defining a block-compressed geometry format that is designed for arbitrary geometry topologies and can be directly consumed by future fixed-function hardware.

CCS Concepts: • **Computing methodologies** → **Graphics systems and interfaces**; **Ray tracing**; **Visibility**; • **Theory of computation** → **Sorting and searching**; **Massively parallel algorithms**.

Additional Key Words and Phrases: bounding volume hierarchy, ray tracing

ACM Reference Format:
Joshua Barczak, Carsten Benthin, and David McAllister. 2024. DGF: A Dense, Hardware-Friendly Geometry Format for Lossily Compressing Meshlets with Arbitrary Topologies. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3, Article 1 (July 2024), 17 pages. https://doi.org/10.1145/3675383

1 INTRODUCTION

The widespread availability of hardware-accelerated ray tracing solutions is gradually altering the real-time graphics landscape. The major graphics APIs, DXR [Microsoft 2020] and Vulkan [Khronos Group 2020] now offer cross-vendor extensions for hardware-accelerated ray tracing that have seen solid adoption by applications. In the same time frame, software innovations on the rasterization front have raised the geometric complexity bar. Epic's Nanite system [Karis et al. 2021] – based on lossily compressed meshlets – enables practical real time rendering of billions of (virtual) triangles.

Authors' addresses: Joshua Barczak, Advanced Micro Devices, Inc., USA; Carsten Benthin, Advanced Micro Devices, Inc., Germany; David McAllister, Advanced Micro Devices, Inc., USA.

However, Nanite's meshlet-based architecture does not compose well with the existing ray tracing interfaces. API-compatible acceleration structure inputs for meshlets do not exist at any point in its pipeline, and conversion to the API formats increases the data size by a large factor. Moreover, current ray tracing hardware uses low-density triangle representations designed for lossless storage of floating-point coordinates. These factors prevent ray tracing-based applications from rendering the same level of geometric complexity as rasterizers.

In this paper, we propose a ray tracing friendly block-compressed data format for dense geometric models named *Dense Geometry Format* (DGF). The format has been designed to support arbitrary geometry topologies and efficient ray tracing. We will address format details, introduce an aggressive tri strip generalization, discuss a reference encoding pipeline, ray tracing performance, and possible hardware and API support. Our format achieves compression rates comparable to Nanite's format, while at the same time providing efficient access to individual triangle data inside the meshlet, which is crucial for efficient ray tracing support.

2 RELATED WORK

Geometry compression has been a research focus for decades. For a general overview we refer to Maglo et al. [2015]. Compressing vertex position data is typically done by quantization [Alliez and Gotsman 2005; Deering 1995; Peng et al. 2005]. For better vertex compaction and crack prevention, Lee et al. [2010] aligned a vertex position to a global grid and subdivided the mesh until vertices inside the smaller meshlets allowed 8-bit quantization. Meyer [2012] dynamically changed the bit rate for vertex encoding in a view-dependent manner.

Epic's Nanite system [Karis et al. 2021] combines several important ideas into a cohesive system. Geometry is represented using clusters or meshlets of up to 128 triangles, and a fine-grained level-of-detail system is used to manage the working set by replacing groups of adjacent high detail clusters with a low-detail equivalent. The set of active clusters is tracked on the GPU by performing a per-frame traversal of the cluster graph using compute shaders. Compressed geometry data is streamed on demand from disk and decoded into a compact in-memory representation, which is then directly rasterized. Vertex positions are tightly packed and stored in variable-precision fixed point. The bit rate is adjustable and can be tuned to the needs of the content, providing a size/quality trade-off. Recently, Kuth et al. [2024] proposed lossy meshlet compression techniques explicitly optimized for rasterization and mesh shaders.

In the context of ray tracing complex geometry, various approaches for compressing geometry data have been proposed. In the following we will briefly discuss the most relevant.

Segovia and Ernst [2010] applied hierarchical mesh quantization combining BVH and triangle data into a single data structure. Vertices within BVH leaves are quantized with respect to the leaves' bounding boxes. Cracks due to different quantization anchors (leaf bounding boxes) are prevented by aligning vertices and leaf bounding boxes to a global fixed point grid.

Tessellation of subdivision surfaces or displaced height fields can often be represented locally by a small grid-like topology. A fixed grid topology does not require adjacency information. This allows for a more compact representation. Several works propose efficient methods for lossy compression of grid-like topologies. For example, Benthin et al. [2021] compress vertex data of small grids by encoding them as offsets from a base primitive, i.e., a bilinear patch.

Benthin and Peters [2023] provide a partial solution to ray tracing Nanite-like lossy compressed meshlets. They demonstrate a cluster-based level-of-detail system for dense geometric models by providing lossily compressed meshlet structures as direct input to the hardware-specific acceleration structure builder. However, this assumes that the builder knows the exact input format, but this is not given in today's ray tracing APIs. Hence, a need for a standard lossily compressed input format that can be directly consumed by API implementations is pointed out.

Recently, hardware support for ray tracing of displaced micro meshes (DMM) [Bickford and Moreton 2023] was introduced. A DMM consists of base triangles with a displacement vector per vertex. These are hierarchically subdivided into micro triangles. The level of hierarchical subdivision is user defined. Displacement vectors are interpolated across the base triangles, scaled by hierarchically encoded scalars, and applied as offsets to the newly created micro vertices. This approach offers a very compact geometry representation and level of detail support. However, the approach cannot represent arbitrary geometry topologies due to the restriction of applying scalar adjustments to modify the interpolated displacement vectors. Models with complex topology, sharp features, or high depth complexity must be represented using a large number of base triangles, which eliminates the advantage. Examples of such problematic models are the *Bicycle* and *Powerplant* model shown in Figure 1. Additionally, the DMM base triangles are the coarsest available simplification of geometry.

3 DENSE GEOMETRY FORMAT (DGF)

3.1 Requirements

As a compressed format for dense geometry data suitable for efficient ray tracing needs to address various requirements, we will discuss those here in more detail.

Topology The format must be able to represent any mesh topology and must not impose any restrictions on connectivity. This also implies that any standard triangle mesh asset must be easily convertible to the format. Additionally, it must be possible to guarantee water-tightness for any kind of connected triangles.

Density Lossy compression of positions is essential to achieve high compression rates. Ideally, the compression rate should be adjustable, so artists can make precise size and quality trade offs for different use cases. Supporting degenerate data containing *INF*, *NAN*, etc, is not a requirement.

Efficient Access The data layout should be block-based, and all data of a given triangle should reside in the same block. This property facilitates efficient parallel decoding and access to individual triangles during ray tracing or shading. Ideally the block size should be chosen with respect to the cache line size of the target hardware to guarantee minimal cache line transfers per block fetch.

Existing API Compliance It is important to comply with existing API specifications, hence the format must be able to encode additional data such as IDs for *primitive* and *object*, as well as an opacity flag per triangle.

Encoding Time Encoding time must be reasonable to allow for quick artist turnaround and conversion times. The encoder should be as simple as possible but at the same time maximize encoding density.

Animated Geometry To facilitate animated geometry, it must be possible to quickly update the vertex positions without breaking the block-based structure.

Many of these properties also apply to existing standards for texture compression like DXT [Waveren 2006] or ASTC [Nystad et al. 2012], for many of the same reasons. In the following we will describe our format, which meets all of the above requirements.

3.2 Format Description

Our dense geometry format (DGF) consists of an array of 128 byte blocks that encode triangle data. Each block holds a maximum of 64 triangles and 64 vertices. The block layout is illustrated in Figure 2. The first 20 bytes are a header that contains vertex anchors (see Section 3.2.1) and encoding

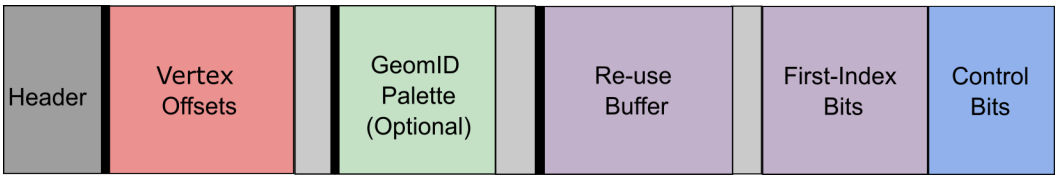


Fig. 2. Layout of a 128 bytes *Dense Geometry Format (DGF)* block. Thick lines indicate byte-aligned boundaries. Light grey regions show unused padding bits. The vertex offsets are used to reconstruct vertex positions (see Section 3.2.1). The geometry ID palette encodes an optional material ID per triangle (see Section 3.2.3). The remaining regions encode the mesh topology (see Section 3.2.2). The control bits and first-index bits are packed from back to front.

parameters for that block. Per-vertex offsets are tightly packed following the header, followed by an optional *Geometry ID palette* (see Section 3.2.3), and a compressed topology encoding (see Section 3.2.2).

3.2.1 Vertex Position Encoding. Floating-point vertices often have excessive precision near the origin, and not enough when further away. This is generally undesirable for geometric models, especially micropolygon geometry. DGF vertices are therefore defined on a 24-bit signed integer grid, inspired by Nanite [Karis et al. 2021]. Each block stores an *anchor* position for each axis (signed, 24b), a per-vertex offset from the anchor (unsigned, 1-16 bit), and a power-of-two scale factor to map from integer coordinates to world coordinates (8-bit IEEE-compliant exponent). Given anchor position A, offset O, and exponent E, a decoded vertex V is given as:

$$V = (A + O) \cdot 2^{E-127}.$$

The intermediate sum is a 25-bit signed integer which is then converted to floating-point. The full range of possible values can be converted exactly. The result is losslessly multiplied by a power-of-two floating-point scale factor. We disallow any encoded value outside the IEEE single-precision float range, and also prohibit denormals, NaNs, and infinite values.

The bit width of the offsets is specified in the block header. An application can trade off between precision and encoding density by adding or removing least significant bits and adjusting the exponent accordingly. We require the sum of the x, y, and z bit widths to be a multiple of 4 in order to minimize muxing cost in a possible hardware decoder. We briefly experimented with parallelogram prediction [Jie et al. 2011], but found it did not benefit our test models (see Section 5), and complicated random access to the vertices.

3.2.2 Topology Encoding. Our topology encoding is a simplification of Deering's more general scheme [Deering 1995]. We use a compressed, generalized triangle strip and allow limited backtracking to increase strip length. The behavior of our strips is illustrated in Figure 3. Each triangle's vertices are ordered such that the edge between the first two vertices is the one that is shared with the predecessor. We store a 2-bit control field per triangle indicating one of the four actions to take to generate triangle i:

- *RESTART(0)*: Consume 3 indices and restart the strip.
- *EDGE1(1)*: Consume 1 index and re-use edge 1 of triangle $i - 1$
- *EDGE2(2)*: Consume 1 index and re-use edge 2 of triangle $i - 1$
- *BACKTRACK(3)*: Consume 1 index and re-use the remaining edge of triangle $i - 2$

The first triangle is always a *RESTART*, and its control bits are omitted. Vertices of re-used edges are swapped on re-use to preserve winding. Mixed winding can be encoded by restarting the strip

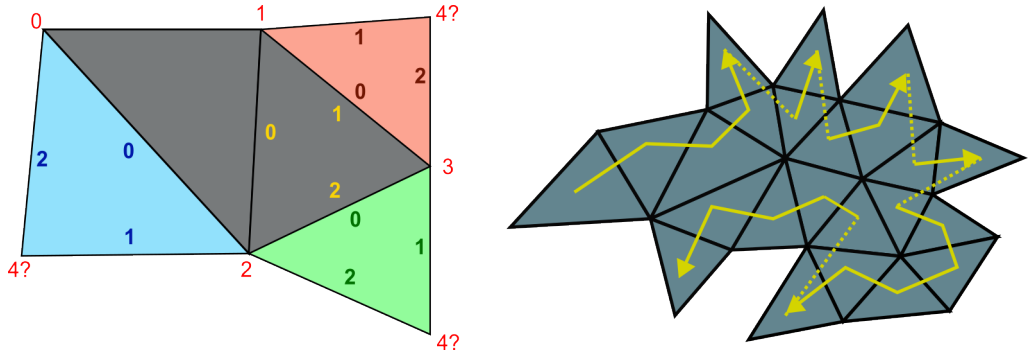


Fig. 3. Generalized backtracking triangle strips. Left: Strip continuation options. Edges of the current triangle are numbered in yellow. Options are *EDGE1* (red), *EDGE2* (green), and *BACKTRACK* (blue). Red labels show vertex indices. Right: A challenging topology that our method can encode in a single strip. The strip path is shown in yellow. Dotted lines indicate backtracks.

on each winding change. *BACKTRACK* is only allowed if the predecessor used *EDGE1* or *EDGE2*. It enables the encoder to handle isolated triangles without restarting the strip (see Figure 3).

In addition to the control bits, we store an index buffer for the strip. We compress the index buffer in the manner of Meyer [2012]. Vertices are ordered by first-use. A bit per index indicates if it is the first reference to its vertex. Vertex offsets are referenced directly on first use, but for subsequent uses are accessed via an indirect index stored in the re-use buffer. The re-use index width is selected on a per-block basis to address the unique vertices and can be set from 3 to 6 bits.

3.2.3 Primitive and Geometry IDs. To comply with the existing ray tracing APIs [Khronos Group 2020; Microsoft 2020] it is also necessary to encode a 29-bit primitive ID, a 24-bit geometry ID, and 1-bit opacity for each triangle. The opacity value determines whether or not any-hit shaders are executed when the triangle is intersected. We encode the primitive IDs implicitly by storing a 29-bit base in the header, and adding the triangle's index in the block. The geometry ID, despite its name, often functions as a material ID. A single, small ID for an entire block is a very common case. It is also common for the *opaque* flag to correlate with material, so we concatenate the two to produce a 25-bit result with the opacity in the lowest bit. We offer two mechanisms for storing this information:

- Constant mode: A 10-bit value is stored in the block header and used by all triangles. This mode is used when all triangles use the same value and the upper bits are zeros.
- Palette mode: A geometry ID *palette* is stored in the block, and the 10-bit header field defines the number of entries (5-bit,1-32), and the number of high-order bits common to all entries (5-bit,0-25).

The layout of the geometry ID palette is illustrated in Figure 4. Its size and position are byte aligned, and it contains the shared most significant bits (MSBs), followed by the set of varying least significant bits (LSBs), followed by an entry index for each triangle in the block. The bit width of

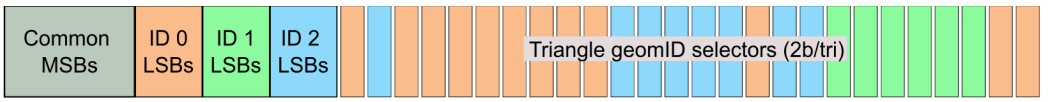


Fig. 4. Example geometry ID palette containing three entries. Each entry stores a 25-bit value (24-bit geomID and 1-bit opacity). High-order bits with the same value across all entries are stored only once. Following the IDs is an array of 2-bit fields that select an ID from the palette for each triangle in the block.

the indices is inferred from the number of entries in the palette. The indices are not present if there is only one entry. This can occur if all triangles use the same ID and its value is too large to use the constant mode.

3.3 Attribute Data

In real-time applications it is common to have vertex or triangle attribute data that depends on the triangle order. Examples might include colors, normals, or index buffers. To achieve low bit rates the DGF encoder requires the ability to reorder the triangles within the mesh, and to rotate triangle vertices in a winding-preserving way. When converting a model to DGF, an application must also post-process any *sideband* data that depends on the input ordering. The widely used methods for post-transform cache optimization [Hoppe 1999; Kerbl et al. 2018; Sander et al. 2007] impose similar requirements. To facilitate this, the encoder produces a remapping table that gives, for each output triangle, the index of the corresponding input triangle, and the position of each vertex in the original vertex order. It also provides a table that holds the original vertex index for each block vertex.

To access per-triangle attribute data, an application can simply use the reconstructed primitive ID stored in the DGF block. Per-vertex data is more nuanced because the same input vertex can appear in multiple blocks. Our approach is to duplicate the shared vertices in each block, and store a per-block offset into the duplicated vertex buffer. If the vertex attributes are large, it is better to keep them in their original order and store the original index for each duplicated vertex. This costs 4 bytes per vertex, but allows the attribute data to be de-duplicated. The crossover point depends on the amount of duplication and the size of the vertex data. Tables 1 and 3 quantify the amount of vertex duplication.

4 DGF GENERATION

Encoding DGF blocks must strike a balance between ray tracing efficiency and encoding density. For ray tracing efficiency (see Section 5.4), it is best to group triangles according to the surface area heuristic (SAH) [Goldsmith and Salmon 1987]. For encoding density, we want to minimize vertex duplication. Optimizing both metrics at once is a challenging problem, so we approximate it by optimizing for SAH at a coarse granularity, and vertex reuse at a finer granularity.

4.1 SAH-Based Clustering

We begin by partitioning the set of triangles into SAH-optimal clusters in the same manner as Benthin and Peters [2023]. We use a binned-SAH algorithm [Wald et al. 2008] with 256 bins and continue partitioning until the triangle count is below a user-specified threshold (we used 128 for all experiments). Using SAH-based clustering enables a fast, cluster-granular BVH build, and improves vertex compression by arranging triangles into compact spatially localized groups.

4.2 Quantization

After SAH-based clustering, the next step is to snap the floating-point vertices to a global integer grid. A single quantization factor is chosen for all triangles, thus preventing cracks. A quantized vertex V_q is computed as:

$$V_q = \text{round}\left(V / 2^e\right).$$

The exponent e is computed as:

$$e = \left\lceil \log_2\left(E_o / 2^{b-1} - 1\right)\right\rceil,$$

where E_o is the maximum edge length of the bounding box over the entire model and b is a target signed bit width. The value of e must be adjusted to avoid overflow in the 24-bit per-block anchors or the 16-bit per-vertex offsets. To prevent offset overflow, the value of e must satisfy:

$$e \geq \left\lceil \log_2 (E_c) - 16 \right\rceil,$$

where E_c is the maximum edge length for any cluster's AABB. To prevent overflow and underflow in the anchor fields, we need two additional constraints:

$$e \geq \left\lceil \log_2 \left(-O_{\min} / \left(2^{23} - 1 \right) \right) \right\rceil,$$

$$e \geq \left\lceil \log_2 \left(O_{\max} / 2^{23} \right) \right\rceil,$$

where $O_{\min}$, $O_{\max}$ are the minimum and maximum vertex coordinate values. The corresponding constraints are ignored if $O_{\min} > 0$ or $O_{\max} < 0$.

The quantization step produces 24-bit integer coordinates for each vertex, and guarantees no more than 16-bit of difference between two vertices in the same cluster. Note that the target precision can be higher than 16-bit as long as triangle sizes are small. The 24-bit inputs are converted to offsets when blocks are formed, and the grouping into SAH-optimized clusters keeps these offsets small.

The exponent selection above can lead to precision loss when the model contains a mix of large and small triangles. In a production pipeline, this can be mitigated by subdividing large triangles, or by isolating and compressing them independently. This is discussed further in Section 5.6.

4.3 Block Packing

After vertex quantization, the next step is to decompose each cluster into one or more DGF blocks. Each cluster is encoded independently, using a greedy algorithm that tries to maximize vertex re-use. The first triangle in the cluster is selected to start a new block, and we search the list for an unused triangle that shares the most vertices with those already in the block. Ties are broken by comparing Morton codes of the triangle centroids and choosing the candidate with lowest Morton code. Once a candidate is found, we attempt to form a block from the current triangle set. If successful, the triangle is retained and we search for another. If unsuccessful, we start a new block, beginning with the unused triangle with lowest Morton code. This process repeats until all triangles are consumed.

The procedure for constructing a block from a triangle set is as follows:

- Compute the set of unique vertices referenced by the triangles.
- Compute the integer axis-aligned bounding boxes (AABB) of the vertices.
- Compute the x, y, z bit widths from the AABB size.
- Scan the geometry IDs and decide whether to use palette or constant mode. If palette mode is selected, compute the size of the palette.
- Construct the triangle strip and compress the triangles index buffer.
- Compute the size of the vertices, topology, and geometry IDs and test the sizes fit.

The encoder retains an intermediate state for the current block, so most of the steps above require only constant work for each new triangle. For example: updating the vertex set simply requires testing 3 entries in a bit vector and adjusting the vertex count and the AABB as required. Triangle strip construction is the exception, and must be repeated each time a new triangle appears.

4.4 Triangle Strip Construction

To construct triangle strips, we use a simple greedy traversal of the triangle adjacency graph (half-edge structure). We build a cluster-level graph once, and use it to quickly build block-level graphs on demand by extracting a subset of the nodes and edges.

Algorithm 1 Decompressing vertex data for the ith triangle in a DGF block. We compute three index buffer addresses by scanning the triangle strip, decode the index value at each address, and reconstruct the corresponding vertices.

$r \leftarrow 1$
$indexAddress \leftarrow [0, 1, 2]$
for $k = 1 \rightarrow index$ **do** ▷ Scan strip from the beginning
  $ctrl \leftarrow Control[k]$
  $prev \leftarrow indexAddress$
  if $ctrl = RESTART$ **then**
    $r \leftarrow r + 1$ ▷ Count restarts
    $indexAddress \leftarrow [2r + i - 2, 2r + i - 1, 2r + i]$
  else if $ctrl = EDGE1$ **then**
    $indexAddress \leftarrow [prev[2], prev[1], 2r + k]$
    $bt \leftarrow prev[0]$
  else if $ctrl = EDGE2$ **then**
    $indexAddress \leftarrow [prev[0], prev[2], 2r + k]$
    $bt \leftarrow prev[1]$
  else if $ctrl = BACKTRACK$ **then**
    if $prevCtrl = EDGE1$ **then**
      $indexAddress \leftarrow [bt, prev[0], 2r + k]$
    else
      $indexAddress \leftarrow [prev[1], bt, 2r + k]$
    end if
  end if
  $prevCtrl \leftarrow ctrl$
end for
for $k = 0 \rightarrow 2$ **do** ▷ Decode indices and vertices
  $vid \leftarrow$CountFirst(FirstIndex,indexAddress[k])
  if $FirstIndex[indexAddress[k]] = 0$ **then**
    $vid \leftarrow ReuseBuffer[indexAddress[k] - vid]$
  end if
  $Vertex[k] \leftarrow (Anchor + Offset[vid]) * Scale$
end for

We specialize our structure for valence 3 to speed up the implementation. For non-manifolds where more than two triangles share the same edge, we discard edges until all nodes have at most 3 neighbors. Non-manifolds are correctly encoded, but extreme cases might cause frequent strip restarts and reduce compression rate.

To construct a strip from a graph, an arbitrary node with minimum valence is chosen and its triangle added to the strip. The node is deleted, and the valences of the neighbors adjusted. The neighbor with minimum valence is added to the strip, deleted from the graph, and the process continues. Whenever a triangle is added, its vertices are rotated to align the first edge to the one connecting it to the strip.

If a triangle has no more neighbors, we attempt to backtrack to the opposite neighbor of the preceding triangle. Backtracking is allowed when the prior triangle has an unused neighbor, and was not a restart or backtrack. If backtracking fails, we restart from an arbitrary minimum-valence node. The combination of backtracking and the minimum valence heuristic is very effective because it causes the strip to walk block boundaries in a spiral pattern, rather than venturing into the interior and becoming trapped. Isolated *ear* triangles, which would otherwise cause restarts, are

	$b = 11$	$b = 12$	$b = 13$	$b = 14$	$b = 15$	$b = 16$	$b = 24$	DMM
Average Geometric Error (%)								
Lowest	0.0314	0.0157	0.0078	0.0039	0.0020	0.001	0.0000	0.001
Highest	0.0812	0.0405	0.0203	0.0101	0.0051	0.0025	0.0006	0.011
Maximum Geometric Error (%)								
Lowest	0.0564	0.0282	0.0141	0.0071	0.0035	0.0018	0.0000	0.1
Highest	0.1458	0.0729	0.0363	0.0182	0.0091	0.0045	0.0012	2.4
Bytes per Triangle								
Lowest	2.74	2.77	2.82	2.87	3.24	3.05	3.68	0.81
Highest	3.30	3.65	4.05	4.37	4.95	5.42	6.95	2.38
DGF Vertex Duplication								
Lowest	1.49	1.51	1.51	1.53	1.54	1.55	1.59	—
Average	1.56	1.57	1.58	1.61	1.65	1.68	1.82	—
Highest	1.85	1.88	1.90	1.92	1.95	2.00	2.16	—

Table 1. Geometric error and memory cost for DGF and DMM. Geometric error is expressed as percentage of AABB diagonal. DGF results use the models from Tables 2 and 3. DMM figures are derived from Figure 19 and Table 1 of [Maggiordomo et al. 2023]. We compute average and maximum error for each model and report the range of results across the model set. Vertex duplication gives the ratio of block vertices to input vertices.

instead our preferred choice, since the strip can detour into them and backtrack back into place, as shown in Figure 3.

4.5 Decompression

Algorithm 1 illustrates the steps to decompress the vertex data of the ith triangle in a DGF block. We scan the strip from the beginning until the desired triangle is found. The address for the third vertex of triangle i is computed by adding 2 for each restart at positions at or before i. If i is a *RESTART*, the second and first vertices decrement this index by 1 and 2, respectively. Otherwise, the index addresses are inferred from the two preceding triangles. This is done by advancing until the desired triangle is reached, retaining four previous index values, and the previous control value. Once the index addresses are known, we retrieve the index values as proposed by Meyer [2012]. If the index at a specified address is the first reference to its vertex, its value is computed by an inclusive prefix sum on the 'is-first' bits. Otherwise, it is extracted from the re-use buffer. The result gives the position of one of the stored vertex offsets. This is then used to reconstruct the floating-point vertex data (see Section 3.2.1).

5 RESULTS

For our evaluation, all pre-processing steps – scene subdivision into DGF meshlets and DGF encoding itself – were run on a CPU (Intel Core I9-13900KF), while for direct ray tracing of DGFs we use a software implementation based on *DXR* intersection shaders [Microsoft 2020]. All rendering tests were conducted on an AMD® Radeon™ 7900 XT GPU and Windows 11.

We start by providing a detailed comparison in terms of compression density, geometric error, and encoding time between DMM and different DGF encoding modes (see Section 5.1). Next, we compare DGF against other alternative compression formats (see Section 5.2), followed by evaluation of using DGF as BVH leaf representation (see Section 5.3). We provide a ray tracing performance comparison between DGF and standard indexed mesh triangles as BVH leaf representation (see Section 5.4). Finally, we provide statistics on triangle strip length and quad formation (Section 4.4).

	Bike	Crytek Sponza	SanMiguel	Sibenik	Rungholt	Powerplant
Triangles (M)	1.67	0.26	9.98	0.075	6.70	12.76
Materials	10	25	287	15	84	66
B/Tri With/Without	3.75/3.73	4.84/4.81	3.47/3.34	5.36/5.17	4.49/4.21	3.05/3.02
Palette	4.98%	8.08%	31.27%	35.48%	51.31%	9.78%

Table 2. Effect of geometry ID compression. *Palette* gives the percentage of blocks using geometry ID palettes. $b = 16$ is used due to the larger spatial extent of these scenes.

5.1 Compression Density and Encoding Time

We first compare geometric error and memory footprint between DGF and DMM (see Table 1). For DGF results, we measure distance between encoded and decoded vertices and express this as a percentage of the AABB diagonal. For DMM, we report the results from Maggiordomo et al. [2023], which measure distance between the input and μ-mesh surfaces. At low bit rates, DGF often has higher error, while at high bit rates, the reverse is true. For the tested scenes, using $b = 14$ roughly matches the average error for DMM. Crucially, DGF error is more stable and predictable. The maximum error is always roughly 1.8× the average, and scales linearly with quantization level, while for DMM the error can vary by as much as 10×. A qualitative analysis is given in Section 5.6.

Table 2 shows the impact of geometry ID compression. We use a selection of OBJ models with different materials and use material ID as geometry ID. The per-triangle IDs tend to be locally similar, and the palette compression de-duplicates them efficiently.

5.2 Comparison to Alternative Methods

Table 3 compares DGF to DMMs [Bickford and Moreton 2023], compressed meshlets [Kuth et al. 2024] and Draco [Google 2017]. Draco is an implementation of EdgeBreaker [Rossignac 1999] and aims at reducing disk footprint and download size. Its sequential representation is not suitable for direct ray tracing, but provides a lower bound on compression rate. The compressed meshlets by Kuth et al. [2024] are designed for efficient decompression by rasterization-based mesh shaders. The format is not ideal for ray tracing because accessing a single triangle requires multiple block reads. The meshlets are also larger than DGF, but their reduced vertex duplication may compensate for this in practice. Compared to DMMs, DGF uses roughly 3× more memory, but in exchange, can handle any input mesh topology, and takes considerably less time for encoding. The majority of the execution time for DGF encoding is spent in repeatedly constructing triangle strips and testing for block fullness.

The reported data rate for Nanite is roughly 9 B/tri in memory and 5.6 B/tri on disk [Karis et al. 2021]. Those figures include level-of-detail hierarchy and vertex attribute data, which makes a direct comparison difficult. However, the position-only numbers in Tables 1 and 3 highlight the competitiveness of DGF.

5.3 BVH Memory Footprint

Table 4 projects the memory savings from using DGF blocks as the leaf triangle format in a BVH. We use the DXR API to obtain compacted BVHs for the input geometry and compute the size of the BVH's (uncompressed) triangle leaf data representation. This is possible because the acceleration structure builder for our target GPU is open-source [AMD 2024a]. DGF reduces the triangle leaf data by 6-10× and total BVH size by 50%. Our target hardware requires uncompressed nodes. BVHs with denser internal node encoding [Liktor and Vaidyanathan 2016; Ylitie et al. 2017] would derive a greater size reduction.

	Telegraph	GW Bust	Murex	Lucy	Statuette	Dragon	Fangyi	Ewer
Triangles (M)	7.94	26.06	3.52	28.06	10.00	7.22	21.57	7.94
Verts (M)	3.97	13.03	1.76	14.03	5.00	3.61	10.79	3.97
Triangle Size (Bytes/Triangle)								
VB/IB	18	18	18	18	18	18	18	18
Draco	0.69	0.91	0.95	0.71	0.85	0.48	0.90	0.94
DMM	0.99	0.81	1.16	0.85	2.38	0.97	1.00	1.29
Meshlets	3.99	4.02	3.98	4.00	3.96	4.00	3.98	3.95
DGF	2.96	3.02	3.29	2.87	3.17	3.24	3.29	3.08
Vertex Duplication Factor								
Meshlets	1.23	1.24	1.22	1.23	1.22	1.23	1.24	1.24
DGF	1.53	1.58	1.58	1.53	1.56	1.58	1.57	1.55
Encoding Time (seconds)								
Draco	2	18	3	9	3	2	6	3
DMM	436	1660	317	1661	563	399	1215	462
DGF	13	46	6	49	17	12	34	13

Table 3. Memory consumption and execution time. DMM figures are based on Table 1 of [Maggiordomo et al. 2023]. Meshlets uses the method of [Kuth et al. 2024] (authors' implementation), with vertex buffer size scaled to approximate $b = 14$. Draco and DGF execution times use $b = 14$ and were measured on an Intel Core I9-13900KF (single-threaded). Draco [Google 2017] is not suitable for direct rendering, but provides a lower bound on compression rate.

Our test hardware stores a maximum of two triangles in a BVH leaf, intersecting them one by one, but many ray tracers have wider triangle processing. For example, Embree [Wald et al. 2014] tests up to 8 triangles at once using vector instructions. In the limit, an implementation might treat entire DGF blocks as BVH leaves and brute-force the intersection tests [Benthin et al. 2021], use ray-space pruning [Benthin et al. 2004], or use a small strip-level structure similar to [Lauterbach et al. 2007]. A full exploration of the design space is beyond the scope of this paper, but we attempt to map it in Table 4 by creating procedural primitives to represent groups of consecutive DGF triangles.

	DXR	DGF	Inner Nodes				DXR	DGF Totals			
	Leaves	Blocks	Δ=2	Δ=4	Δ=8	Δ=64	Total	Δ=2	Δ=4	Δ=8	Δ=64
Bike	34.22	3.75	39.51	16.84	8.68	1.69	73.73	43.27	20.59	12.44	5.45
Sponza	34.24	4.84	28.59	11.81	6.26	1.61	62.83	33.43	16.65	11.10	6.45
San Miguel	35.78	3.47	41.51	15.89	8.19	1.49	77.29	44.98	19.35	11.66	4.96
Sibenik	34.24	5.36	29.46	12.33	6.56	1.74	63.70	34.82	17.69	11.92	7.11
Rungholt	32.00	4.49	22.90	10.80	5.77	1.36	54.90	27.40	15.30	10.26	5.85
Powerplant	34.72	3.64	41.36	14.38	7.48	1.35	76.08	45.01	18.03	11.12	5.00

Table 4. Projected acceleration structure size for a set of models (bytes per triangle). *DXR Leaves* is the size of the triangle data for the DXR BVH. *DGF Blocks* is the triangle data size for DGF using $b = 16$. *Inner Nodes* columns give the size of internal nodes and meta-data for a given maximum triangle count per leaf (Δ). *DGF Totals* is the sum of DGF block size and internal nodes for a given leaf size.

5.4 Rendering

We implement DGF ray intersection by using procedural primitives to invoke an intersection shader for each triangle. The shader performs on-the-fly format decompression into 32-bit floating-point numbers followed by ray triangle intersections in software. We compare this against a

		Cache Requests(M) (Hit %)					
	VALU Util.	Instruction 32KB	L0 16KB	L1 256KB	L2 4MB	L2 Miss (Millions)	Time (ms)
Telegraph (7.9M Triangles, 242K DMM Base Triangles)							
Reference	50%	682 (100)	316 (49)	162 (14)	138 (32)	93 (1.00×)	9.39 (1.00×)
DGF ($b = 16$)	89%	1431 (100)	411 (67)	134 (21)	105 (42)	61 (0.65×)	17.97 (1.89×)
DMM Proxy	66%	692 (100)	322 (64)	114 (28)	81 (71)	23 (0.24×)	6.79 (0.72×)
Crytek Sponza (0.26M Triangles, 40K DMM Base Triangles)							
Reference	68%	1483 (100)	838 (51)	406 (33)	271 (79)	56 (1.00×)	14.30 (1.00×)
DGF ($b = 16$)	94%	2875 (100)	1350 (74)	345 (46)	187 (81)	35 (0.63×)	34.09 (2.38×)
DMM Proxy	74%	1907 (100)	945 (62)	347 (38)	212 (94)	11 (0.19×)	13.90 (0.97×)

Table 5. Statistics for ray traversal of two scenes at 1080p. Scenes are rendered with simple diffuse path tracing, 3 samples per pixel, up to 3 bounces per sample. We used the Radeon™ GPU Profiler [AMD 2024b] to collect statistics. DGF decoding has up to 2.4× higher cost, but reduces the L2 misses (memory bandwidth) by 40-50%. DMM further reduces L2 miss rate since its BVH is built over a small set of coarse base triangles.

reference intersection shader that retrieves data from a conventional vertex and index buffer (see Table 5). Comparing software tests ensures that the BVH topology and ray traversal loop are as similar as possible between the two scenarios. We approximate DMM by testing against the base triangles, using the AABBs of the extruded prism. Each triangle fetches a dummy displacement block containing zeros. This approach underestimates the computational cost of DMM but produces comparable memory traffic.

For the reference shader, we use one procedural primitive for each input triangle. For DGF, we supply 64 procedural primitives per DGF block and use degenerate AABBs for the unused slots. These are removed by the BVH builder. This allows our intersection shader to obtain the DGF block ID from the primitive ID MSBs and the triangle index from the LSBs. The GPU driver constructs 4-wide (uncompressed) internal BVH nodes over these input primitives, each 128 bytes in size.

The DGF intersection test performs many additional loads to retrieve data from various parts of the block. The control bits and first-index bits are all loaded, placed in registers and then parsed to determine the index buffer addresses. The extra memory reads are absorbed by the first-level cache, but the ALU cost of scanning the triangle strip reduces performance by a factor of up to 2.3×. Importantly, misses in the last-level cache are sharply reduced, because the compact triangle representation reduces the working set size and thereby frees up cache space for more BVH nodes. As in section 5.3, an architecture with a more compact BVH node can be expected to derive a larger benefit. The DMM implementation traverses a smaller BVH, and can offer better performance as long as the content can be well represented (see section 5.6).

5.5 Strips and Quads

Table 6 compares the generalized triangle strip builder described in section 4.4 to competing methods that build traditional strips. For each DGF block we decode its triangle list, and construct strips using the alternative method. The DGF strip builder achieves the highest average strip length due to its use of generalized backtracking strips.

Several ray tracing architectures [Intel Corporation 2024] [Simon Fenney 2024] natively intersect pairs of edge-adjacent triangles, so we also report the probability that consecutive triangles in a DGF block have a shared edge between them. This occurs over 90% of the time.

	Telegraph	GW Bust	Murex	Lucy	Statuette	Dragon	Fangyi	Ewer
				Average Strip Length				
Zeux	10.00	8.04	8.54	9.24	8.93	9.73	8.86	8.96
Tunneling	13.52	11.93	11.61	12.22	12.04	13.45	12.10	11.80
DGF ($b = 16$)	19.32	14.81	15.42	17.82	16.13	18.06	16.38	16.25
Quad Rate	91.80%	90.72%	90.85%	91.50%	91.20%	91.51%	91.27%	91.25%

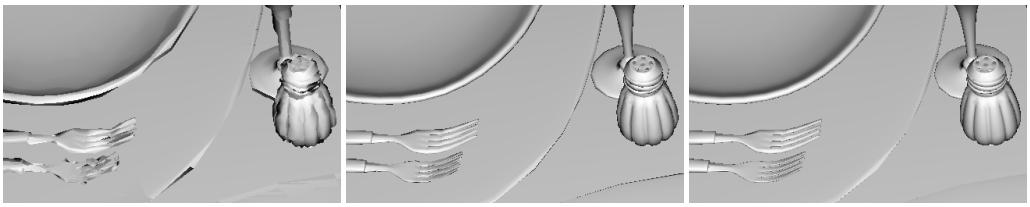
Table 6. Comparison of triangle strip lengths produced by various methods. *Zeux* uses the method of [Evans et al. 1996] as implemented by [Kapoulkine 2024]. *Tunneling* uses our implementation of [Stewart 2001]. Our strip builder achieves the highest strip length. *Quad Rate* is the probability that two consecutive triangles in a block share an edge. Our use of strips enables good utilization rates for quad-based intersection pipelines.

5.6 Qualitative Error Analysis

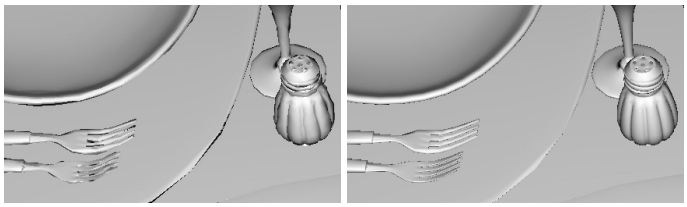
DMM is a displacement mapping technique, and automatic base mesh generation does not always capture high-frequency detail, as shown in Figure 6. These errors could be mitigated in practice through manual tuning of the base mesh, but this is labor-intensive. Other potential problem cases for DMM are detailed in [Nvidia 2023]. In this case, DGF needs no manual intervention to produce an acceptable result.

Figure 5 shows a failure case for DGF. As discussed in Section 4.2, the encoder can lose precision if the input contains a mix of large and small triangles. The San Miguel scene contains large backdrops that are visible through doorways. These cause the encoder to choose a lower quantization factor to avoid offset overflow, which results in inadequate precision for the high-polygon tableware. The resulting errors have a low magnitude (relative to scene size), but are problematic nonetheless. In this case, visible artifacts are eliminated simply by encoding with $b = 24$, which increases the byte per triangle ratio by 18% (see Figure 5b).

Since a problematic triangle can be arbitrarily large, we also evaluated other mitigations at $b = 16$. Figure 5d shows the result of encoding the scene with the backdrops removed. This improves accuracy, and has the counter-intuitive effect of reducing the compression rate, since the vertices use larger offsets. Another strategy (Figure 5e) is compressing triangle groups in isolation



(a) $b = 16$ (3.47B/tri) (b) $b = 24$ (4.12B/tri) (c) Uncompressed (19.7B/tri)



(d) $b = 16$ (3.77B/tri) (e) $b = 16$ (5.51B/tri)

Fig. 5. Compression errors in the San Miguel scene, with mitigations. (5a) Direct encoding, ($b = 16$). (5b) Direct encoding ($b = 24$). (5c) Uncompressed reference. (5d) $b = 16$ with large tris removed. (5e) $b = 16$ compressing individual triangle groups in isolation.

and concatenating the blocks. This eliminates the artifacts, but hurts compression rate even more, by creating larger numbers of low density clusters.

Another potential failure case is a large, elongated mesh with large coordinate values on one axis and small ones on the others. In this case the only option is to divide the mesh into locally encoded pieces and use instance transforms to position them. This is likely to align with best-practice anyway, since large, monolithic BLAS can degrade TLAS quality.

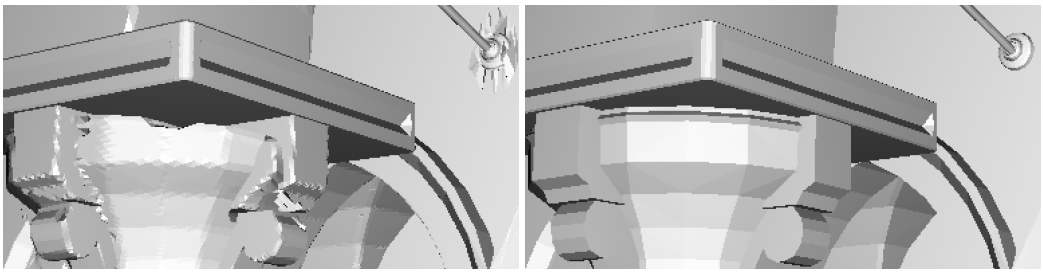


Fig. 6. DMM failure case. Left: A DMM representation of the Sponza scene (40K base tris, 16M μ-tris, 16MB), created using the toolchain of [Maggiordomo et al. 2023]. DMM struggles with sharp corners and small, high-frequency features. Right: DGF encoding of the original mesh with $b = 16$ (262K tris, 1.23MB).

6 EXTENSIONS

In the following we will discuss possible hardware and API support for DGF and outline an extension for dynamic content.

6.1 Hardware Support

The DGF decoding cost (see Section 5.4) can be remedied by deploying dedicated decoding hardware. For example, a hardware strip-scan unit could be built that executes an unrolled strip decoding loop, computing index buffer addresses in parallel, and passing them through a chain of conditional swaps until the desired triangle is reached. If a large strip cannot be fully decoded in one cycle, multiple smaller decoders could be chained in a pipeline. The result could be returned to a shader or used as input for additional decoding logic. Overall, the hardware investment would be well justified by DGF's reduction in memory footprint (see Table 3) and memory bandwidth (see Table 5).

6.2 Dynamic Geometry

Full DGF encoding is too expensive to run per frame, but the simple linear vertex structure facilitates fast vertex update. To support animation, DGF blocks can be built for a model's rest pose, with the encoder modified to *lock* the vertex offsets at a particular, high bit width, instead of choosing a tight fit to the input data. This reduces density by $\sim 2 - 4\times$ (see Table 7), but it enables the coordinate values to expand and contract as necessary. An application can use a compute shader to evaluate the vertex animation, quantize the results, and pack new vertex positions into the blocks, while leaving topology and geometry IDs intact. Locked DGF blocks are less compact than the compressed meshlets of [Kuth et al. 2024], but retain the important property that triangle data can be accessed using a single memory fetch. DMM trivially enables animation of the base mesh, but only to the extent that a static, scalar displacement from a moving base surface can provide an acceptable result. When the mesh is subjected to complex deformations such as skinning this will often not be the case.

	Telegraph	GW Bust	Murex	Lucy	Statuette	Dragon	Fangyi	Ewer
Bytes Per Triangle								
DGF ($b = 14$)	2.96	3.02	3.29	2.87	3.17	3.24	3.29	3.08
Locked 12b	6.28	6.45	6.36	6.38	6.37	6.37	6.31	6.37
Locked 16b	8.66	8.77	8.71	8.77	8.73	8.78	8.67	8.71
Ratios								
Locked 12b	2.12×	3.18×	1.93×	2.22×	2.01×	1.97×	1.92×	2.07×
Locked 16b	2.92×	4.32×	2.65×	3.05×	2.76×	2.71×	2.63×	2.83×

Table 7. Impact of locked offset width for update-able blocks. Using locked offsets reduces density by $\sim 2-4\times$, but enables the vertex positions to change. With locked offsets, DGF offers $\sim 2-3\times$ higher density than standard indexed meshes, but $\sim 1.5-2\times$ lower than the compressed meshlets of [Kuth et al. 2024] (see Table 3.

6.3 API support

For maximum benefit, DGF must be standardized by the major graphics APIs [Apple 2023; Khronos Group 2020; Microsoft 2020]. This could by done by extending the acceleration structure build APIs to accept an array of pre-computed DGF blocks as input, just as they currently accept pre-compressed textures. An implementation would copy the input blocks and construct vendor-specific internal nodes above them. Implementation-specific post-processing, such as quad formation, might be applied at this stage. The total triangle count for the blocks would also be supplied, since implementations might need to reference triangles individually. Existing shader intrinsic functions that provide access to primitive and geometry IDs would return the values encoded in the input. To facilitate access to vertex attributes (see Section 3.3), the application could optionally specify a 32bit meta-data field for each block to indicate the location of its attributes. Additional intrinsic functions could retrieve this meta-data and the block-local vertex indices for a hit triangle.

7 CONCLUSION AND FUTURE WORK

We propose *DGF*, a block-based format for efficiently storing dense geometry data. Its design and structure have been optimized for ray tracing specific use cases. Although it offers a lower density than competing displacement-based lossy compression approaches, it is able to support any mesh topology, and offers higher density than other meshlet-based lossy compression formats. The ability to quickly encode any topology allows for an easier integration into existing asset creation pipelines. Even though software-based intersection of DGFs is currently slower than a simpler uncompressed leaf data representation, future direct decoding support in hardware would completely offset this current limitation, and unlock substantial memory footprint and bandwidth reductions. In the future, we are interested in simulating different variants of direct hardware decoding support for DGF and in improving the encoding algorithm.

ACKNOWLEDGEMENT

Model courtesy: Reef Crab (threedscans.com), Bike (Yasutoshi Mori), Powerplant (University of North Carolina). The authors would like to thank Quirin Meyer, Holger Gruen, and the anonymous reviewers for their feedback. Max Oberberger provided meshlet data for Table 3. The authors would like to acknowledge Andrew Kensler, Trevor Hedstrom, and Mohammed Al-Obaidi for significant contributions to DGF.

REFERENCES

Pierre Alliez and Craig Gotsman. 2005. Recent Advances in Compression of 3D Meshes. (2005), 3–26.

AMD. 2024a. *GPU Raytracing Library.* Retrieved April 22, 2024 from https://github.com/GPUOpen-Drivers/gpurt

AMD. 2024b. *Radeon GPU Profiler.* Retrieved April 22, 2024 from https://gpuopen.com/rgp/

Apple. 2023. Metal Documentation. https://developer.apple.com/documentation/metal

Carsten Benthin and Christoph Peters. 2023. Real-Time Ray Tracing of Micro-Poly Geometry with Hierarchical Level of Detail. *Computer Graphics Forum* 42, 8 (2023).

Carsten Benthin, Karthik Vaidyanathan, and Sven Woop. 2021. Ray Tracing Lossy Compressed Grid Primitives. In *Eurographics 2021 - Short Papers.*

Carsten Benthin, Ingo Wald, and Philipp Slusallek. 2004. Interactive ray tracing of free-form surfaces. In *Proceedings of the 3rd International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa* (Stellenbosch, South Africa) *(AFRIGRAPH '04).* Association for Computing Machinery, New York, NY, USA, 99–106. https://doi.org/10.1145/1029949.1029968

Neil Bickford and Henry Moreton. 2023. Getting Started with Compressed Micro-Meshes. In *NVIDIA GPU Technology Conference.* https://register.nvidia.com/flow/nvidia/gtcspring2023/attendeeportal/page/sessioncatalog/session/1666430278669001BFSR

Michael Deering. 1995. Geometry compression. (1995), 13–20.

F. Evans, S. Skiena, and Amitabh Varshney. 1996. Optimizing triangle strips for fast rendering. In *Proceedings of Visualization.* 319 – 326.

Jeffrey Goldsmith and John Salmon. 1987. Automatic Creation of Object Hierarchies for Ray Tracing. *Computer Graphics and Applications* 7, 5 (1987), 14–20.

Google. 2017. Draco Geometry Compression. https://google.github.io/draco/

Hugues Hoppe. 1999. Optimization of mesh locality for transparent vertex caching. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99).* 269–276.

Intel Corporation. 2024. *Intel® Arc™ Graphics Developer Guide for Real-Time Ray Tracing in Games.* https://www.intel.com/content/www/us/en/developer/articles/guide/real-time-ray-tracing-in-games.html

Xu Jie, Jiang Hao, and Li Zhen. 2011. 3D Mesh Compression by Generalized Parallelogram Predictive Vector Quantization. *Information Technology Journal* 10 (2011).

Arseny Kapoulkine. 2024. MeshOptimizer. https://github.com/zeux/meshoptimizer

Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. A Deep Dive into Nanite Virtualized Geometry. https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf in Advances in Real-Time Rendering in Games: Part I (proc. SIGGRAPH courses).

Bernhard Kerbl, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. 2018. Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the modern GPU. *Proc. ACM Comput. Graph. Interact. Tech.*, Article 29 (2018).

Khronos Group. 2020. Vulkan Ray Tracing Extensions Specification. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_ray_tracing.html

Bastian Kuth, Max Oberberger, Felix Kawala, Sander Reitter, Sebastian Michel, Matthäus Chajdas, and Quirin Meyer. 2024. Towards Practical Meshlet Compression. arXiv:2404.06359 [cs.GR]

Christian Lauterbach, Sung-Eui Yoon, and Dinesh Manocha. 2007. Ray-Strips: A Compact Mesh Representation for Interactive Ray Tracing. In *2007 IEEE Symposium on Interactive Ray Tracing.* 19–26. https://doi.org/10.1109/RT.2007.4342586

J. Lee, S. Choe, , and S. Lee. 2010. Compression of 3d mesh geometry and vertex attributes for mobile graphics. 09 (2010).

G. Liktor and K. Vaidyanathan. 2016. Bandwidth-efficient BVH Layout for Incremental Hardware Traversal. In *Proceedings of High-Performance Graphics.* 51–61.

Andrea Maggiordomo, Henry Moreton, and Marco Tarini. 2023. Micro-Mesh Construction. *ACM Trans. Graph.* 42, 4, Article 121 (jul 2023), 18 pages. https://doi.org/10.1145/3592440

Adrien Maglo, Guillaume Lavoué, Florent Dupont, and Céline Hudelot. 2015. 3D Mesh Compression: Survey, Comparisons, and Emerging Trends. *ACM Comput. Surv.* 47, 3, Article 44 (2015).

Quirin Meyer. 2012. Real-Time Geometry Decompression on Graphics Hardware. (2012).

Microsoft. 2020. DirectX Raytracing (DXR) Functional Spec. https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html

Nvidia. 2023. *Nvidia Gameworks DMM Toolkit.* https://github.com/NVIDIAGameWorks/Displacement-MicroMap-Toolkit/blob/main/docs/asset_pipeline.md

J Nystad, A Lassen, A Pomianowski, S Ellis, and T Olson. 2012. Adaptive Scalable Texture Compression. *High-Performance Graphics 2012, HPG 2012 - ACM SIGGRAPH / Eurographics Symposium Proceedings* (2012).

Jingliang Peng, Chang-Su Kim, and C. C. Jay Kuo. 2005. Technologies for 3D mesh compression: A survey. *J. Vis. Comun. Image Represent.* 16, 6 (2005), 688–733.

Jarek Rossignac. 1999. Edgebreaker: Connectivity Compression for Triangle Meshes. *IEEE Transactions on Visualization and Computer Graphics* (1999).

Pedro V. Sander, Diego Nehab, and Joshua Barczak. 2007. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Trans. Graph.* 26, 3 (jul 2007), 89–es. https://doi.org/10.1145/1276377.1276489

Benjamin Segovia and Manfred Ernst. 2010. Memory Efficient Ray Tracing with Hierarchical Mesh Quantization. In *Graphics Interface*. 153–160.

Simon Fenney. 2024. *Hot3D: Ray Tracing With Imagination.* https://www.highperformancegraphics.org/slides23/2023-06-_HPG_IMG_RayTracing_2.pdf

A. James Stewart. 2001. Tunneling for Triangle Strips in Continuous Level-of-Detail Meshes. In *Graphics Interface*. 91–100.

Ingo Wald, Carsten Benthin, and Solomon Boulos. 2008. Getting Rid of Packets - Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs. In *Symposium on Interactive Ray Tracing*. 49–57.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics* 33 (2014).

J.M.P. Waveren. 2006. Real-Time DXT Compression. (05 2006).

Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of High Performance Graphics* (Los Angeles, California) *(HPG '17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3105762.3105773