DOI: 10.1111/cgf.15002

COMPUTER GRAPHICS *forum*
Volume 43 (2024), number 1, e15002

End-to-End Compressed Meshlet Rendering

D. Mlakar,[1] M. Steinberger[1] and D. Schmalstieg[2]

[1]Graz University of Technology, Graz, Austria
{daniel.mlakar, steinberger}@icg.tugraz.at
[2]University of Stuttgart, Stuttgart, Germany
dieter.schmalstieg@visus.uni-stuttgart.de

Abstract

In this paper, we study rendering of end-to-end compressed triangle meshes using modern GPU techniques, in particular, mesh shaders. Our approach allows us to keep unstructured triangle meshes in GPU memory in compressed form and decompress them in shader code just in time for rasterization. Typical previous approaches use a compressed mesh format only for persistent storage and streaming, but must decompress it into GPU memory before submitting it to rendering. In contrast, our approach uses an identical compressed format in both storage and GPU memory. Hence, our compression method effectively reduces the in-memory requirements of huge triangular meshes and avoids any waiting times on streaming geometry induced by the need for a decompression stage on the CPU. End-to-end compression also means that scenes with more geometric detail than previously possible can be made fully resident in GPU memory. Our approach is based on a novel decomposition of meshes into meshlets, i.e. *disjoint primitive groups that are compressed individually. Decompression using a mesh shader allows de facto random access on the primitive level, which is important for applications such as selective streaming and fine-grained visibility computation. We compare our approach to multiple commonly used compressed meshlet formats in terms of required memory and rendering times. The results imply that our approach reduces the required CPU–GPU memory bandwidth, a frequent bottleneck in out-of-core rendering.*

Keywords: modeling, compression algorithms

1. Introduction

Open-world games, scientific simulations and many other graphics applications can involve huge scenes that are challenging to store, transmit and render. If the scene does not fit into GPU memory in its entirety, the application needs to schedule data streaming and decompression, such that the data become available just in time. For such *out-of-core* rendering, an optimal compression rate must be weighed against in-memory footprint and GPU friendliness.

Conventionally, a rendering system expects the scene to be in-core, *i.e.* fully resident in GPU memory. For scenes larger than memory, a *sparse* representation with sequential access enables out-of-core processing in the sense that a spatially bounded portion of the scene is loaded into memory (and decompressed, if any compression has been applied). Such sequential access patterns can be applied to tasks such as segmentation, smoothing or simplification, but they are typically not compatible with rendering—draw calls require *direct access* to the geometric primitives. If the scene is structured into multiple meshes representing individual objects, one can simply select which meshes to load. However, the scene may consist of a single huge mesh or several large meshes, which cannot be fully resident. In this case, a common method is to segment the meshes into regions. Regions should have a finite spatial extent, and their primitive count should be chosen in a GPU-friendly manner. Ideally, a region corresponds to a meaningful part of an object [CG08, CKLL09], which can be loaded and decompressed individually.

The requirements of streaming rendering are not new [TMJ98, LH04], but are still highly relevant in contemporary game technology. Streaming is prominently addressed in the Microsoft DirectStorage API [Yeu20], which bypasses the CPU by streaming directly from non-volatile storage into GPU memory, and has recently been extended to include hardware-assisted Lempel-Ziv-like decompression [Hoe22, CH22, Ura22]. A similar approach is available in Epic's *Nanite* [Epi20] rendering engine.

However, the aforementioned tools for out-of-core rendering pipelines do not directly aim at preserving GPU memory. Rendering is assumed to work from an uncompressed representation, in the

sense that shader input can be read directly from memory. Since both GPU memory size and bandwidth must be considered scarce resources, we prefer *end-to-end compressed rendering*, which decompresses triangles directly in the shader and limits the GPU memory requirements to only the compressed, rather than the uncompressed, geometry representation. Such an approach avoids the need for a separate decompression stage on the CPU and considerably reduces the overall memory footprint of the geometry data. It also simplifies the engine code, since there is no need to maintain separate compressed and uncompressed representations.

In this paper, we explore a compressed rendering infrastructure that performs decompression directly in a *mesh shader* [Kub18], operating on groups of geometric primitives called *meshlets*. The ability to decompress geometry on the fly in the mesh shader makes it easy to design a streaming system, since the same compressed representation can be kept in GPU memory and in non-volatile storage. To support decompression in the mesh shader, we present connectivity representations that lend themselves to fully parallel processing. In summary, we make the following contributions:

- We present the first complete end-to-end compressed mesh rendering pipeline.
- We introduce a novel decomposition of meshes into disjoint primitive subsets (called *laced wires*), which allow random access without vertex duplication.
- We describe how to create laced wires and how to efficiently decode them in parallel on the GPU.
- We compare our approach to multiple compressed and uncompressed mesh representations.

2. Related Work

This paper focuses on triangle meshes, which are the most widely used mesh representation in computer graphics. The most common data structure, an indexed triangle mesh, consists of a triangle list storing three consecutive vertex indices per triangle and a vertex list storing vertex data, *e.g.* positions or normals. While this data structure can be directly processed by the graphics pipeline, it is not very compact. Hence, there is a large body of work on mesh representations that require less memory through compression of vertices, connectivity or both [MLDH15]. Most single-rate compression methods for meshes combine a lossless encoding of triangle connectivity with a lossy encoding of vertex data. For brevity, we do not consider lossy connectivity compression, *i.e.* mesh simplification or re-meshing, but rather assume that the geometry quality is already given at the desired rate.

2.1. Mesh compression

Vertex and attribute compression. Lossy compression of vertex data is acceptable in many applications, since the vertex coordinates and other attributes usually do not require full floating point precision [Dee95, Cal01, MSGS11, HV01]. The application of quantization by converting coordinates to a fixed point format and stripping the least significant bits is often possible without any visual deterioration [PBCK05, KXW*18]. In fact, some meshes, such as those acquired with 3D scanning, tend to store only noise in their least significant bits. However, hand-modelled meshes may have their parts expressed with varying precision, making a mechanism for non-uniform quantization necessary. A straightforward method to address non-uniformity is hierarchical decomposition of the mesh into parts with homogeneous precision inside each part. Compression rates can potentially be further increased by exploiting local coherence in the value sequence through delta-encoding or prediction [IA02]. In contrast to position quantization, domain and precision requirements of other frequently used attributes, like normals [CDE*14] and texture coordinates, are known in advance, making their quantization less challenging.

Connectivity compression. Most compact connectivity representations use some form of constrained graph traversal of the mesh and encode the traversal steps using a set of symbols with low entropy. A popular choice are triangle strips [Ise00], given as a sequence of vertices, where each new vertex together with its two predecessors in the sequence defines a new triangle. While standard triangle strips attach each new vertex to alternating edges of the previous triangle, a special symbol is used to choose the edge in generalized triangle strips to allow longer strips. With more complex traversal patterns, *e.g.* aligning strips next to each other [Cho97] or in concentric circles [BPZ99], encoding efficiency can be further improved.

Other traversal patterns include spanning trees [Tau98] or region growing [Ros99], the latter probably being the most influential idea in connectivity compression. In region growing, the border of the already encoded part of the mesh is incrementally extended, guided by a custom traversal strategy, and newly encountered triangles are encoded. Noteworthy traversal strategies include the cut-border machine [GS98] as well as EdgeBreaker [Ros99] and its many improvements [CR04, SKR01]. Alternatively, region growing can also be expressed in terms of the valence of added vertices [TG98], *e.g.* in FreeLence [KPRW05].

The Laced Ring (LR) [GLRL11] structure can be seen as a kind of dual representation of a generalized triangle strip. A very long triangle strip stores hardly more than one vertex reference for each encoded triangle. However, in practice, the additional operations required for a generalized triangle strip (edge swap, restart) incur a 30–40% overhead [GLRL11]. The laced ring replaces triangle sequences with sequences of adjacent vertices. For each edge between two vertices, two references to laces, *i.e.* triangles formed by extra vertices adjacent on each side of an edge, are stored. While this representation also requires approximately one reference per triangle, it is easier to find long vertex sequences than to find long triangle sequences (a triangle has three neighbours, while a vertex has six on average [GLLR13]). We expand upon the idea of laces in this paper.

Encoding shared boundaries. If a large mesh has to be segmented into regions before compression, one has to deal with shared boundaries between the regions. Boundaries must be handled with care, to avoid cracks between regions. The most common approach to avoid cracks is to split meshes along the edges, leading to shared vertices between regions. In this case, shared vertices are duplicated across regions, which increases the memory footprint and may increase the effort to keep vertex replicas consistent. Duplication is avoided if the vertices are stored only for one region and referenced in all other regions [YL07, CG08], but this requires reference counting

and separate vertex loading. Finally, the border vertices can also be stored separately as vertex sequences (*wires*). The wires can be created from an existing subdivision into regions [CKLL09], or the regions can be created by recursively subdividing the mesh by placing wires [CH09]. In the latter case, we can traverse the hierarchical sub-division to extract any desired region size.

2.2. Compressed domain rendering on the GPU

The idea of on-the-fly decompression on the GPU has previously been explored for domains in which large datasets are common. Examples of such domains are volume rendering [SW03, BCF03, FM07], rendering of large terrain data [DSW09, LC10] or spatial acceleration structures [LH07]. All these domains deal with special input formats involving samples organized in lattices, grids or boxes.

In contrast, the connectivity of a triangular mesh is usually given as a general graph, which requires more complex methods for decompression when using single-instruction multiple-data (SIMD) parallelism. Data dependencies of typical low-entropy encoding schemes, where each step relies on the previous one, are not well suited for SIMD. Instead, a semi-regular layout is required that allows many threads direct access to compressed data. Such a layout can take the form of long strips [JBG17] or boundaries in region-growing schemes [MSGS11].

Unfortunately, data-parallel decompression in these methods requires scattered writes of the decompressed data to GPU memory. Consequently, rendering involves a separate pass, which consumes substantial memory bandwidth. In addition, the overall size of the mesh is limited to what fits in memory. Commercial game engines, such as Unreal Nanite [KSW21], strike a trade-off between compression and speed by decompressing into an intermediary, low-entropy (but randomly accessible) format in GPU memory.

A recent paper by Nikolaev *et al.* [NFR22] investigates an idea similar to ours by comparing the use of geometry shaders and mesh shaders to decode and render on the fly. However, unlike us, they employ a region-growing scheme [Ros99], which has severe drawbacks for GPU operation: Not only does region growing preclude locality of data references, but the decoding scheme (region growing of mesh connectivity and delta encoding of geometry) is also inherently serial. Hence, the authors can support parallel decoding only on the level of meshlets, but not on the level of individual triangles. Unfortunately, this significant limitation is not discussed or evaluated in the paper. We demonstrate the performance improvements of our approach over the work of Nikolaev et al. in Section 5.

2.3. Mesh layout optimization

In standard rendering with depth buffer hardware, primitives can be submitted to the graphics pipeline in any order, which lets us freely choose the best *mesh layout* for a given purpose. Locality of references is often a primary concern, since it has several benefits. First and foremost, it improves the coherence of the vertex cache on the GPU, reducing redundant per-vertex computations [Dee95, Hop99]. Many algorithms have been proposed to produce either cache-aware [LY06, CK07] or cache-oblivious [YLPM05] layouts. Cache aware algorithms require exact knowledge of cache sizes, while cache-oblivious algorithms optimize locality without details about the cache. For example, the matrix of triangle-to-vertex references can be sorted for minimal bandwidth [IL05]. Even simple techniques such as following a space-filling curve can yield layouts that are within a few percent of the theoretical optimum [VSSP12]. Besides reference locality, other criteria can be considered in layout generation, such as compact memory footprint, geometric coherence or minimal overdraw [CSN*12, SNB07, HS16].

The mesh layout is also implicitly considered when a compression algorithm traverses a mesh during encoding. Depth first traversals, which produce long sequences of connected triangles for compression can be similar to cache-oblivious algorithms, if a 'winding' path is used [CK07]. However, non-winding sequences may compress equally well [GLLR13], while having a rather poor cache coherence [DBGP05]. Likewise, breadth-first traversals, which use region growing [Ros99], refer to the entire mesh and are not necessarily cache coherent. In other words, good compression does not imply good cache coherence.

Moreover, a modern GPU operates on *batches*, *i.e.* primitive groups with an upper bound on the number of vertices and triangles. Optimal throughput is contingent not only on the reference location, but also on a batch-friendly layout [YL06]. Batches can be computed by greedy splitting of a cache-oblivious layout, *e.g.* to minimize overdraw at runtime [SNB07]. However, if the mesh is already segmented during preprocessing, *e.g.* to generate a compressed mesh representation that supports random access, we would like to choose segments that also serve as GPU-friendly batches.

3. Compressed Meshlet Representation

We propose *laced wires*, a random-accessible, compressed mesh structure that combines the advantages of previous methods, namely laces [GLRL11] and wires [CKLL09, CH09]. Our representation is both compact *and* SIMD-friendly, which is critical for end-to-end compression with high SIMD utilization. A wire is simply a sequence of vertices connected by edges [CKLL09]. We use wires to represent the boundary and the interior of a meshlet, such that two adjacent meshlets refer to the same wire at their shared boundary. Laces store the vertices of the two triangles to the left and right of a given edge. We apply laces to the wires so that neighbouring laces wires interlock without redundant vertices. LR uses ring-shaped vertex sequences that are made as long and winding as possible. The LR scheme is optimized for global storage use in CPU memory, at the expense of giving up any notion of local references. Our goal is to fetch and decode a limited amount of memory into the cache of a SIMD unit on the GPU to describe a meshlet in its entirety. Therefore, our representation consists of multiple short sequences per meshlet, rather than a single long one as in LR. The interlocking of neighbouring laces avoids duplicated vertices, making the format free of redundant storage and precludes any cracks between triangles. See Figure 1 for a comparison of LR and laced wires.

The construction of our compressed representation comprises four steps: *Meshlet construction* (Section 3.1) assigns triangles to meshlets. *Connectivity encoding* (Section 3.2) converts the meshlet's triangle data into the laced wire format. *Vertex encoding* (Section 3.3) finds the minimum bitrate per vertex in a local coordinate

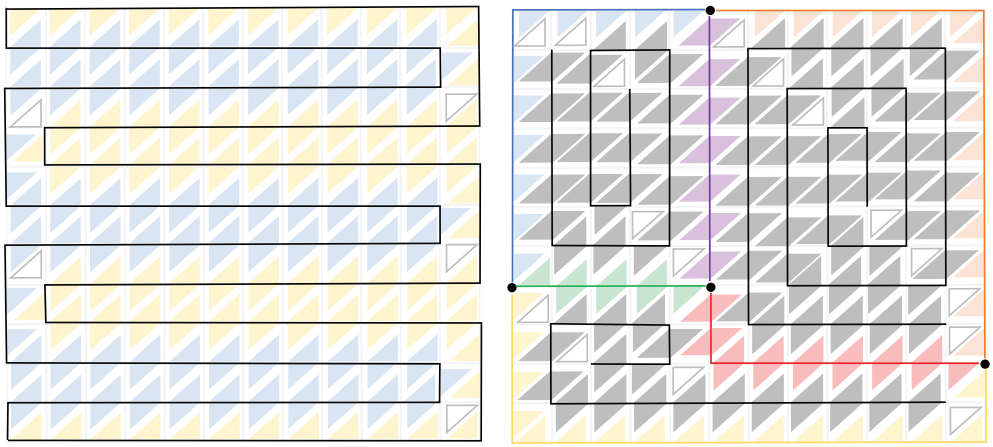

Figure 1: *(Left) The Laced Ring (LR) consists of a single, closed sequence of vertices (shown in black). Triangles attached to edges of the LR sequence are shown in blue and yellow only for better discrimination. Some isolated triangles (shown as gray outlines) remain; most of these triangles can be integrated without extra storage requirements using special codes. (Right) The same mesh decomposed into three meshlets (top left, top right, bottom), delineated by six external wires shown as coloured polylines (blue, orange, green, red, yellow, magenta). The three internal wires (shown as black polylines) are greedily constructed using a spiral-like Hamiltonian path. Adjacent triangles stored with a wire are displayed in the same colour as their wire. Corner vertices are shown as black dots.*

frame considering a user-defined error tolerances. Packing the data into a *binary format (Section 3.4)* completes the construction.

3.1. Meshlet construction

We cluster triangles into mutually disjoint meshlets, such that the primitive limits are not exceeded: We allow for a maximum of 84 triangles and a maximum of 64 vertices, as recommended by Kubisch [Kub18]. Ideally, a meshlet is a connected, compact and flat sub-mesh. We expect that, if a meshlet is flat, the ‘‘vertical’ dimension of its geometry can be better compressed, and a meshlet with compact shape will exhibit better locality of references overall.

Our meshlet construction algorithm is based on hierarchical face clustering using a quadric error metric [GWH01]. Starting from singletons, *i.e*. clusters each containing only one triangle, clusters are merged iteratively. The approach operates on a dual representation of the mesh with clusters as nodes and weighted edges connecting nodes that correspond to adjacent clusters. Clusters are only considered adjacent if they share at least one edge. The collapse of a dual edge corresponds to the merging of two adjacent clusters. After each merge, we delete all incident dual edges on the new cluster that would lead to clusters exceeding the maximum number of vertices or triangles when collapsed. We terminate the meshlet generation when no more legal merging operations of any two adjacent clusters are possible, *i.e*. once there are no more dual edges in the dual-mesh.

3.2. Connectivity encoding

To encode the connectivity information of a meshlet, we introduce *laced wires*. We adopt the idea of ‘laces’ [GLRL11], *i.e*. specifying triangles adjacent to an edge sequence rather than a triangle sequence. We apply laces to wires of vertices shared by two neighbouring meshlets and call this structure an *external wire*. The interior of each meshlet is also converted into a laced wire; the resulting structure is called an *internal wire*.

Each meshlet with k neighbours consists of one interior wire and k exterior wires. In Figure 1, left, an example of LR is shown, which consists of a single vertex cycle. On the right of the figure, our data structure is shown: The mesh is decomposed into three meshlets, leading to six external wires (in color) and three internal wires (in gray).

Unlike a decomposition of the mesh into meshlets, which have duplicated vertices along the boundary, the laced wire representation is free of redundancies. Each triangle is stored in exactly one wire; each vertex is stored in exactly one wire, except for the *corner* vertices shared by three or more meshlets. However, the geometry of a meshlet can be entirely reconstructed from the wires associated with the meshlet, making random access possible.

The order of vertices in a wire follows a sequence of edges. In external wires, the orientation of the edge sequence is set counterclockwise along the boundary of the adjacent meshlet with the smaller index. For an internal wire, a Hamiltonian path is determined using a greedy algorithm that resembles an erosion process (Figure 1, right). If no Hamiltonian path exists, it is approximated by multiple paths chosen to be as long as possible. To find the Hamiltonian path, vertices are sorted into levels based on their shortest path to the boundary. A path is constructed by iteratively adding a new vertex connected by an edge to the current tip of the path, always prioritizing vertices in lower levels. This strategy conquers the meshlet interior from the boundary in a spiral-like pattern. If none of the remaining free vertices is connected to the tip of the path, any free vertex can be connected through an intermediary ‘virtual’ edge by inserting a special symbol that signals a sequence restart.

Figure 2: *Triangle configurations distinguished in decoding. From left to right: regular triangle requiring one reference triangle, corner triangle requiring two references, unconnected triangle requiring three references, adjacent two/three reference triangle pair.*

The laces of external wires refer to vertices in adjacent internal wires and vice versa. External wires store references to vertices in internal wires using the native numbering of the adjacent internal wire on either side. Conversely, internal wires must refer to several external wires. Random access requires that each reference to a vertex of an external wire must be resolvable without iterating over the entire boundary. Thus, we define a unique ordering of boundary vertices, starting at the corner shared with the neighbouring meshlet with the smallest index and commencing in counterclockwise order. References to external wires use this ordering and are resolved using a parallel index rewriting step based on the sequence of corners (and, hence, external wires) along the boundary.

In an optimal path, each triangle is adjacent to exactly one edge of the path and can be encoded as a 1-reference triangle (1RT). Unfortunately, optimal paths can rarely be found, so we have to handle two additional configurations [GLRL11], as shown in Figure 2. First, a two-reference triangle (2RT) adjacent to two edges, must not be encoded twice; therefore, we have to invalidate one of its references. Second, a triangle that is not adjacent to any edge cannot be omitted and therefore must be encoded as a 3-reference triangle (3RT). All 3RT are assigned to an internal wire; *i.e.* no irregular triangles can be present in an external wire. To reduce the number of references, a 3RT and a 2RT which share the non-path edge of the 2RT can be encoded together using the two references of the 2RT.

3.3. Geometry encoding

We encode vertices of each wire by uniformly quantizing them in a per-wire local coordinate frame. The local coordinate frame directly determines the quantization domain. Therefore, we try to identify a quantization domain that minimizes the number of bits required per vertex, as described below.

Per-wire local coordinate frame. First, we construct the coordinate frame for each wire. As our meshlets are constructed with planarity in mind, we can find a tightly fitting oriented bounding box (OBB) by aligning it with the dominant plane of the meshlet. The OBB specification is further quantized to nine values (rotation, translation and scale for each dimension) with one byte each.

Vertex quantization. We use the OBB of a wire to compute the actual quantization of the vertex positions. Our quantization scheme allows choosing the desired precision at the granularity of individual wires, which can be important for faithfully representing meshes with small details, or for supporting multiple level of detail representations per object. Care is taken to ensure that the full vertex precision can be recovered from the quantized vertices (see the supplement for details). Corner vertices, *i.e.* vertices shared by more than two meshlets, are encoded in the coordinate frame of the entire mesh and are stored separately as a pseudo-wire (without connectivity). Note that each vertex is stored exactly once, making cracks impossible even after heavy quantization.

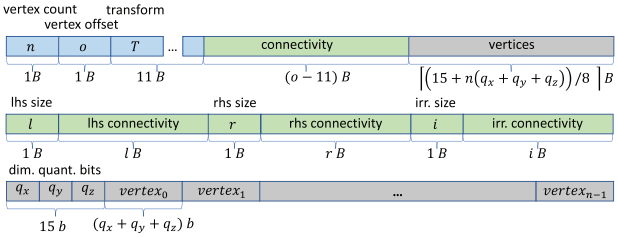

Figure 3: *Binary representation for laced wires. The header (*blue*) contains the number of vertices n as well as an offset o to the vertices in the binary representation and the local wire transformation T. The connectivity data (*green*) is split into sections corresponding to left-hand side, right-hand side and irregular connectivity, each preceded with the corresponding size. Vertex data (*gray*) contain 15 bits for quantization bits per dimension followed by the quantized vertices.*

3.4. Binary format

Each laced wire is represented as a fully self-contained bit string containing all the information required for decoding after random access. We distinguish between header data, connectivity data and vertex data (Figure 3). The header contains the number of vertices and the OBB. The connectivity data stores the vertex references for the left- and right-hand side laces in separate sub-sequences, as only one of them is required to decode an external wire for a meshlet. Each symbol is one byte in size, with two bits encoding the triangle type and six bits distinguishing up to 64 vertices. The vertex data (quantized values x, y, z values) is stored in a packed bit string.

In addition to the encoded wires, we need meta-data to look up the wires for each meshlet: we need the indices of its internal wire, its external wires, and its corners. We store this information in a per-meshlet structure, together with an offset to the meshlet's first external wire index.

4. Decoder Implementation on the GPU

We pre-fetch compressed meshlet data from non-volatile storage as needed. Unlike conventional preloading of geometry, the pre-fetched data are stored in GPU memory in compressed form, thereby significantly reducing the memory footprint. By combining the gains in memory efficiency resulting from (1) sparse residency of scene data and (2) the compressed in-memory representation, huge scenes can be handled that would not be possible with either method alone.

For decoding, we use a pipeline consisting of an *amplification shader* followed by a *mesh shader* as shown in Figure 4. The former is used to decide which meshlets need to be decompressed in a subsequent mesh shader for rendering. Mesh shaders allow us to re-define the geometry stage of the GPU pipeline in a completely

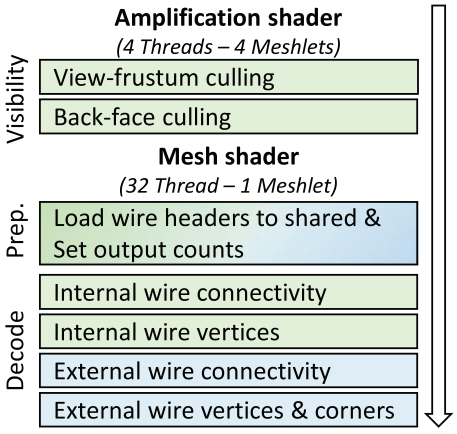

Figure 4: *In this overview, colours indicate whether internal wire data (*green*) or external wire data (*blue*) or both are involved. Each amplification shader instance is launched with a group of four threads, each deciding visibility for a single meshlet. Depending on the visibility, 0–4 mesh shader instances are launched. Each mesh shader instance is executed with 32 threads and decodes a single meshlet from its internal wire and the external wires.*

free manner. We show how the mesh shader can be employed for compressed rendering, such that both the input and the output of the geometry stage use memory bandwidth frugally. Input bandwidth is preserved by loading only compressed meshlets. The final decompression into triangles, vertices and attributes happens on the fly in the mesh shader. Therefore, the full bandwidth for the uncompressed mesh stream is only required between the mesh shader and the rasterizer stage, where it is supported by hardware caches.

4.1. Amplification shader

The amplification shader is used to select the meshes required to render the current frame. Cluster-based culling techniques are well known and let us discard more geometry earlier in the pipeline [KMGL99, HA15, Wih16]. The first step applies view frustum culling. We use the local wire transformation of the internal wire to define a bounding sphere of the meshlet. We can use the centre of the bounding box as the centre of the bounding sphere and half the bounding box diagonal as an initial estimate of the sphere's radius. Since the bounding volume is tested very frequently, we found that a bounding sphere is faster than a bounding box, as it requires transforming only one centre-point instead of eight corners. We use 8 bits per meshlet to store a scale for the initial radius, to ensure that the scaled internal wire bounding sphere contains the entire meshlet. To perform frustum culling, we transform the centre and scaled radius from wire local coordinates to the global coordinate frame and test it against the six frustum planes.

A second step applies back-face culling, using a cone that covers all normals of the meshlet. Since the meshlets are already very flat by construction, most normals will point approximately in the direction of the local x-axis. Therefore, instead of explicitly storing a cone axis, we use the x-axis of the local coordinate frame of the internal wire. We use the centre of the internal wire bounding box as the cone apex, similar as before for the bounding sphere, and we use 8 bits per meshlet to store the cone angle, such that it contains all triangle normals of the meshlet. An angle exceeding 90° implies a strongly curved meshlet, which should be split into multiple meshlets. To ensure that the apex is in the negative half-space of all triangles, we use another 8 bits to store an offset of the apex from the centre along the cone axis (Figure 5).

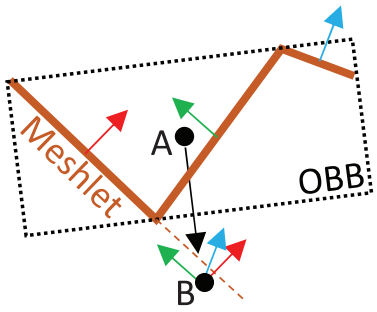

Figure 5: *For backface culling, an oriented bounding box (dashed rectangle) enclosing the meshlet triangles (brown polyline) is computed. The angle of a cone enclosing the normals (red, green and blue vectors) is determined. Moreover, the cone apex B is displaced from the bounding box centre A so that it lies in the negative half-space of all triangles.*

With the local transformation of the internal wire, we can perform frustum and back-face culling, with only 3 bytes additional data per meshlet for the bounding sphere radius scale, the cone angle and the apex offset. If a meshlet is not culled, a mesh shader instance is launched to decompress it.

4.2. Mesh shader

Each mesh shader instance is launched with a thread group (32 threads) and is responsible for decoding a single meshlet. The vertices of internal and external wires have to be dequantized and transformed from the wire's local coordinate frame into the global one. Furthermore, triangle indices have to be decoded from the laced wire representation of the internal wire and from one side of each involved external wire. The mesh shader requires specifying the number of output vertices and triangles prior to any writing to the output arrays. Therefore, we traverse all the wires (internal and external) and load their headers into shared memory. We extract the total number of vertices from the headers and pass it to the mesh shader. Unfortunately, the number of triangles in the meshlet cannot be determined in the same manner, as it cannot be directly inferred from the number of symbols that describe a wire's connectivity (which includes irregularities, such as empty references and restarts). Therefore, we always set the number of mesh shader output triangles to the maximum number of triangles, *i.e.* to 84, and pad all unused triangles with zeros. We empirically determined that zero-padding is faster than explicitly storing and filling in the exact number of triangles.

Internal wire connectivity. For internal wires, we must decode both sides of the wire connectivity and the irregular triangles. We

assign one thread to each symbol on either side of the wire. Each thread reads one symbol and determines whether it encodes a triangle. Thread voting forms a bit mask that indicates which triangles exist. By counting the bits set in the mask, each thread can determine its write offset in the output array. Threads that have encountered a valid triangle decode their vertices and write the result to the output array. Each thread then reads a new symbol until the input is consumed. Finally, irregular 3RT connectivity is decoded with one thread per triangle.

Internal wire vertices. Internal wire vertices are decoded in parallel with one thread per vertex. First, each thread determines the quantization bits per dimension, which can be found in the first 15 bits of the vertex data. From the vertex size, each thread computes its offset into the encoded wire vertices and unpacks the quantized value for each dimension into a register. From there, the vertex is dequantized to get the position in the wire's local coordinates. The transformation is unpacked from the header that resides in shared memory, and the vertex is transformed into the global coordinate frame. After applying the view-projection transformation, the vertex is written to the output.

External wire connectivity. Decoding the connectivity of external wires is similar to internal wires, but must consider multiple external wires instead of one single internal wire. Each external wire contains a different number of symbols, making it more challenging to decode the symbols in parallel. Instead of parallelizing over the edges of a single wire as before, we parallelize over multiple wires. Decoding of external wires commences in two stages. In the first stage, we set up the parallel decoding work. We iterate over the external wires and determine which side of the external connectivity we have to decode. For each symbol of each external wire, a thread is reserved. In the second stage, symbols are decoded in parallel, until no more symbols are left to decode. As before, threads vote if they have encountered a valid triangle to form a bit mask. The bit mask lets the threads write the decoded triangle to the right output position. In the unlikely case that we cannot assign all symbols of a single wire to threads, we decode the wire's connectivity in multiple iterations.

External wire vertices. The decoding of an external wire vertex follows the same procedure as that used for internal wire vertices. The quantized values are unpacked from their binary representation, dequantized into positions given in the local coordinate frame and subsequently transformed into global coordinates using the transformation stored in the wire header. A single external wire usually does not have enough vertices to utilize all threads of the mesh shader's thread group. Therefore, we parallelize over the vertices of multiple wires with different numbers of vertices. Joint decoding of multiple wires requires a prefix sum over the number of vertices in each wire in order to obtain an offset into the vertex output array. During the prefix sum, each thread determines the work description for one vertex, *i.e*. the offset of the external wire assigned to the thread, together with the wire index. We then iterate over the wires and assign threads to vertices, until a thread group is fully assigned. Next, we use register shuffles to pass the work description to the decoding thread. Each decoding thread computes the vertex size of its wire, finds the offset to its vertex, decodes it and writes it into the output array at the location implied by the offset plus the vertex index inside the wire. If the wire is on the right-hand side of the meshlet, the vertex order is reversed to follow the meshlet boundary. The thread group goes on to decode the next batch of vertices, until all wires have been decoded.

Two successive external wires that form the boundary of a meshlet share a corner. Therefore, we can use one thread per external wire to retrieve the starting corner and write it to the corresponding position in the output array. Note that only the corners of the currently processed mesh need to be loaded. Moreover, all corner vertices are encoded in the same global coordinate frame, so their transformation can be stored once in constant memory.

5. Evaluation

We evaluate our approach in terms of space (achieved compression rates) and time (decoding performance) compared to other compressed and uncompressed mesh formats.

5.1. Meshlet formats

To investigate the influence of meshlet geometry and connectivity compression, we compare our laced wire format with four alternative representations, each configured with two different internal formats (basic and compact), resulting in a total of six candidates for comparison. All basic versions use full-precision 32 bits vertex coordinates and 32 bits primitive indices.

Triangle soup meshlet (TS). Arguably, the most trivial format is to represent each meshlet as a triangle soup, where each triangle has a copy of each of its vertices. Although this format results in the least compact representation, no decoding is required. Among the meshlet formats, this format serves as a baseline to better understand the trade-off between data size and decoding effort. To load a meshlet, only the offset into the global buffer and the number of primitives are required. Note that, despite its simplicity, TS represents a meshlet format, due to the local grouping of primitives. We did not compare a global triangle soup format, since we deemed it too unsophisticated. Instead, we compare with a global indexed mesh (see below).

In the compact version of this format, the vertices are quantized globally according to the size of the overall mesh bounding box. The quantized vertices can subsequently be packed in memory such that each vertex requires the exact number of bits to which it is quantized. The meshlet record is padded to the next 32-bit word boundary.

Indexed meshlet (IX). The IX format avoids redundant copies of a vertex inside a meshlet, as present in TS, by using index buffers. IX relies on a global vertex buffer to store unique mesh vertices. Meshlets contain a vertex index buffer that stores references in the global vertex buffer for each vertex referenced in the meshlet. In addition, each meshlet has a primitive index buffer that defines the triangles locally, *i.e*. by referencing the local vertex index buffer. This two-level indirection from primitives to local vertex indices and further on to global vertices improves compactness.

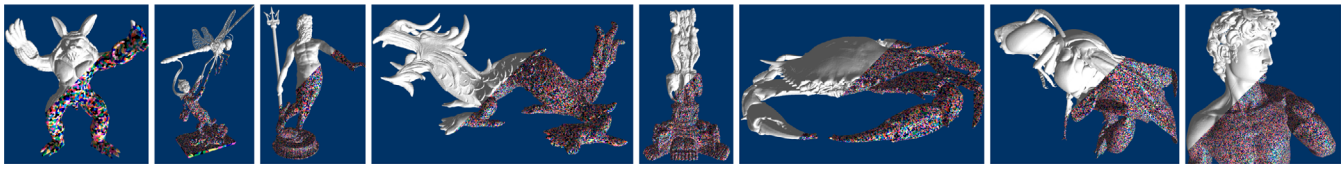

Figure 6: *Our test data comprise three scenes composed of the large meshes shown here. From left to right: Armadillo, Dragonfly, Neptune, Dragon, Statuette, Crab (*Female Blue Crab c/o The Smithsonian*), Bee (*Eulaema Meriana Bee c/o The Smithsonian*), David (*c/o The Digital Michelangelo Project*). In the bottom, right half of each image, the meshlets are indicated by random colour coding.*

In the compact version of this format, we quantize the vertices according to the overall mesh bounding box as before and pack them in memory. Furthermore, we store each reference from a primitive to the vertex index buffer in only 8 bits, since we only need to distinguish 64 or fewer vertices. The entries in the vertex index buffer are stored in 32 bits, so the vertices of large meshes are addressable.

Self-contained meshlet (SC). This format introduces some vertex redundancy to make meshlets independent and self-contained. In contrast to IX, the SC format does not have a local vertex index buffer, but, instead, each meshlet contains its own vertices directly. Vertices at the boundary are duplicated. To access its data, each SC stores an offset from the start of its primitive indices and its vertices, as well as the number of triangles and vertices in the meshlet.

For the compact version, the vertices are quantized and packed again, and the meshlet triangles are represented in 3 bytes as in IX.

Global indexed mesh (GI). In addition to the different meshlet representations, we also added numbers for a GI, the most widely used and commonly known format. The GI consists of a global vertex buffer and a global index buffer that defines the triangles by referencing the vertex buffer accordingly.

For connectivity, the GI requires 12 bytes per triangle. Packing of indices is not possible because a large vertex buffer needs to be addressed; hence, no compression can be achieved. However, vertices can be stored either in a basic format (12 bytes per vertex) or in a globally quantized format, as described in Section 5.2.

5.2. Evaluation setup

All experiments were carried out on an Intel Core i7-7700 equipped with 64 GB of RAM and an NVIDIA RTX 4090. We used three different scenes with different sizes, labelled S, M and L, which are comprised of a variety of different meshes, as shown in Figure 6. Scene L contains all meshes, while scenes M and S are subsets, as detailed in Table 1. All formats use the same decomposition of meshes into meshlets and differ only in how the meshlets are represented.

For the vertices of the input meshes (before encoding), we empirically determined a global quantization that leads to visually pleasing results for all alternative formats, and our approach was configured to undercut the resulting maximum decoding error. Most meshes were quantized to 14 bits per vertex coordinate, except for *Armadillo* (11 bits), *Dragonfly* (13 bits) and *David* (16 bits).

Table 1: *General information about meshes in our test scenes. The columns for scene S, M and L indicate if a particular mesh is included in the corresponding scene. Furthermore, the number of triangles n_t, vertices n_v and meshlets n_m in the mesh and the average number of triangles $\overline{n_t}$ and vertices $\overline{n_v}$ per meshlet are given.*

	Scene			Tri	Vert	Mlets	Tri	Vert
	S	*M*	*L*	n_t $\times 10^6$	n_v $\times 10^6$	n_m $\times 10^3$	$\overline{n_t}$	$\overline{n_v}$
Armadillo		×	×	0.3	0.2	5.4	63.5	46.4
Dragonfly		×	×	2.7	1.3	41.9	63.3	47.2
Neptune	×	×	×	4.0	2.0	63.1	63.5	46.9
Dragon	×	×	×	7.2	3.6	113.0	63.9	47.4
Statuette	×	×	×	10.0	5.0	157.6	63.5	46.9
Crab		×	×	11.3	5.7	177.8	63.6	46.8
Bee		×	×	16.9	8.5	265.4	63.9	47.0
David			×	56.2	28.2	887.7	63.3	46.3
Scene S				21.2	10.6	333.7		
Scene M				52.5	26.2	824.3		
Scene L				108.7	54.4	1711.9		

5.3. Compression rates and decoding times

In order to evaluate compression rates, we compare the required memory footprint of our approach with the alternative formats (TS, IX, SC and GI) in the basic and compact versions. Memory requirements and per-frame times are presented in Table 2. The compression rates achieved by the compact versions over the basic versions are 2.23× smaller for TS, 2.38× for IX, 2.91× for SC and 1.22× for GI. Except for TS, where the applied optimizations almost double the average frame time over all scenes (0.56× of the framerate), they do not cause a significant change in per frame time (1.13× for IX, 0.99× for SC and 1.00× for GI).

We attribute the large impact of the compact format on frame time in TS to the fact that almost all the work required for decoding is concerned with vertices, since the connectivity is implicit in this format, while substantially more vertices have to be decoded than in all the other formats. Therefore, unpacking and dequantizing the vertices introduce a substantial decoding overhead. Moreover, memory transfers increase as vertex coordinates are generally no longer aligned to 4 byte addresses. Note that the other approaches manage to amortize the decoding work by reducing the transfer from GPU memory, and SC even performs slightly better.

Table 2: *Required space (s_b, s_c) and per-frame render time (t_b, t_c) for different scene sizes using the basic and compact formats in comparison to laced wires. Additionally, we show the compression rate (s_b/s_c) and relative time (t_b/t_c) of the compact over the basic format. Subscript b denotes the basic, and c, the compact version.*

	Basic s_b MB	Compact s_c MB	Rate s_b/s_c	Basic t_b ms	Compact t_c ms	Rate t_b/t_c
			Scene S			
TS	767	338	2.27	2.08	3.68	0.56
IX	450	188	2.39	0.74	0.71	1.04
SC	448	153	2.93	0.66	0.67	0.99
GI	382	310	1.23	1.93	1.93	1.00
Laced wires		95			1.60	
			Scene M			
TS	1896	831	2.29	5.13	9.16	0.56
IX	1113	464	2.40	2.13	1.77	1.20
SC	1108	376	2.95	1.62	1.62	1.00
GI	945	767	1.23	6.47	6.47	1.00
Laced wires		236			3.92	
			Scene L			
TS	3927	1850	2.13	10.69	19.30	0.55
IX	2304	981	2.35	4.32	3.77	1.15
SC	2290	807	2.84	3.30	3.32	0.99
GI	1957	1611	1.21	10.44	10.46	1.00
Laced wires		509			8.18	

Table 3: *Details on scene partitioning in our approach. For internal and external wires, we list the number of wires w_i, w_e, number of triangles t_i, t_e, number of vertices v_i, v_e, and the number of corners c. The subscript i denotes internal wires, and, e, external wires.*

	Int. wires			Ext. wires			Corners
	w_i $\times 10^6$	t_i $\times 10^6$	v_i $\times 10^6$	w_e $\times 10^6$	t_e $\times 10^6$	v_e $\times 10^6$	c $\times 10^6$
Scene S	0.34	12.18	6.19	1.00	9.04	3.76	0.66
Scene M	0.82	30.21	15.40	2.46	22.28	9.20	1.64
Scene L	1.71	63.63	32.41	5.10	45.07	18.60	3.41

Comparing absolute decoding times of the compact versions in Table 2 reveals that among the methods competing with laced wires, SC achieves both the best compression rate and the fastest decompression time. SC encodes the 109 M triangles and the 54 M vertices of the large scene in 807 MB and renders a frame in 3.3 ms. By comparison, GI requires about 3× longer in both its basic and compact versions.

The laced wires method beats all competitor formats in terms of compression rate, only requiring 63% of the next-best compressing competitor format, SC. The additional compression rate comes at the price of higher decoding cost: It takes 8.18 ms to decode and render the large scene, which is about 2.5× the frame time of SC, but still 28% faster than the standard format GI. To better understand how the required memory in our approach is distributed, we show a detailed breakdown of the three scenes in Tables 3 and 4. Internal and external wires contribute approximately equal to the data

Table 4: *Data for internal wires (d_i), external wires (d_e) and corners (d_c), total data required to decode and render the scene (d_t). Padding to 32 bits boundaries (typically $1-4\%$ overhead) is included in the data. Adjacency data (d_a) stores external wire identifiers for each internal wire and two corner indices for each external wire.*

	d_i MB	d_e MB	d_c MB	d_a MB	d_t MB
Scene S	36.81	37.55	3.48	17.27	95.12
Scene M	91.75	93.23	8.52	42.60	236.10
Scene L	203.68	197.83	18.92	88.37	508.81

Table 5: *Memory consumption for connectivity and geometry of the different approaches in their original (s_b) and compact form (s_c), the relative compression rate of compact over basic (s_b/s_c) and the compression rate of compact over the baseline provided by the compact global indexed mesh (g_c/s_c). Furthermore, we list the memory per element for basic and compact formats (e_b, e_c), where an element is a triangle for connectivity and a vertex for geometry.*

	Basic s_b MB	Compact s_c MB	Rate s_b/s_c	Rate g_c/s_c	Basic e_b B	Compact e_c B
			Connectivity			
GI	1304	1304	1.00	1.00	12.00	12.00
TS	0	0			0	0
IX	1318	340	3.88	3.84	12.13	3.13
SC	1318	340	3.88	3.84	12.13	3.13
Laced wires		219		5.96		2.01
			Geometry			
GI	653	306	2.13	1.00	12.00	5.63
TS	3927	1848	2.12	0.17	72.16	33.97
IX	986	639	1.54	0.48	18.12	11.74
SC	972	463	2.10	0.66	17.86	8.51
Laced wires		280		1.09		5.14

size and together constitute around 80% of the total required memory. The corner data only require approximately 3–4%, although the vertices are quantized in a single large domain and therefore require more bits to achieve the requested precision. Adjacency information, *i.e.* the data that associates external wires to the internal wire of a meshlet and two corners with each external wire, makes up about 18% of the total amount of data.

5.4. Compression ratio of connectivity

We now analyse the required memory for connectivity separately, as shown in Table 5. For brevity, we only investigate the large scene in this analysis.

TS does not explicitly require space for connectivity. The original IX and SC approaches are quite similar to GI in how they define connectivity using primitive indices, but additionally require the number of triangles as well as an offset into the primitive indices per meshlet. In contrast to GI, we can reduce the primitive indices in IX and SC to 8 bits, as they refer to the vertices locally, resulting in a compression ratio of 3.88.

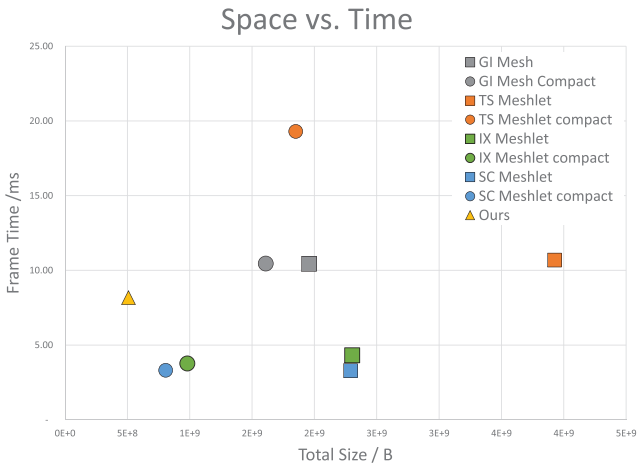

Figure 7: *All compared mesh and meshlet formats for comparison in a space versus time diagram. Basic versions as squares, compact versions as circles, ours as triangle.*

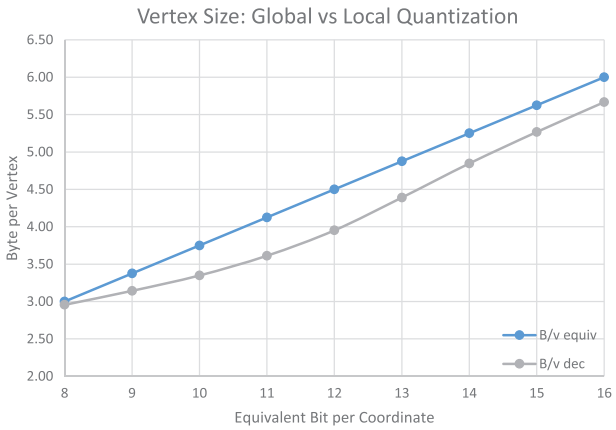

Figure 8: *Comparison of vertex size in our approach* B/v dec *when matching maximum decoding error of a global quantization* B/v equiv.

For laced wires, the memory requirements come from encoding the references of the laces. Our approach requires data that cannot be explicitly assigned to geometry or connectivity, *i.e.* the adjacency data. We split this data into two portions relative to the number of triangles n_t and the number of vertices n_v in the scene and assign them to connectivity and geometry, respectively. $n_t/(n_t+n_v)$ of the general data is added to the connectivity.

To establish a common baseline, we compare the compression ratios relative to the connectivity of GI. We can see that IX and SC reach a compression ratio of 3.84×, which translates to 3.13 bytes per triangle, compared to 12.13 bytes in their respective basic versions and 12 bytes in GI. Our approach has a connectivity compression ratio of 5.96× compared to GI, which is reflected in a triangle size of 2.01 bytes. This result means that laced wire connectivity compresses 1.56× better than IX and SC.

5.5. Compression ratio of geometry

GI requires less memory for geometry than the meshlet formats (except laced wires). The reasons are that TS duplicates vertices in each triangle, IX requires a local vertex index array per meshlet with one 4 bytes index in the global vertex array, and SC contains a copy of vertices at the boundary of the meshlet for each adjacent meshlet.

The memory for geometry in laced wires is computed as everything required to decode the vertices,*i.e.* the offset to the first vertex byte in the wire, the number of vertices and the quantization bits used for each dimension of the meshlet vertices. Furthermore, it contains 9 bytes for the local transformation per wire.

Laced wires take the lead with a compression ratio of 1.09× compared to GI with quantized vertices. Hence, laced wires are the only method that could actually reduce the per-vertex size compared to the compact GI format. Adding additional vertex attributes that can take advantage of the local meshlet coordinate frame would further help amortize the 9 bytes overhead per meshlet caused by the local meshlet transformation. Similarly to the connectivity data, $n_v/(n_t+n_v)$ of the general data that cannot be explicitly assigned to connectivity or geometry is added to the geometry data.

TS takes the last spot, as its redundant vertices induce a large overhead. IX and SC cannot amortize the additional data (local indices and copied boundary vertices), resulting in compression rates of 0.48× and 0.66×, respectively. Compared to 5.63 bytes per vertex in GI, these compression rates result in vertices with 11.74 bytes for IX, 8.51 bytes for SC and 5.14 bytes for our approach.

Figure 7 shows the approaches evaluated in a space/time diagram. Laced wires achieve the smallest memory requirements. IX and SC can be decoded and rendered faster, but require significantly more memory. TS is not viable, as it requires a significant amount of time and memory. GI as a standard format is outperformed by SC and laced wires in both space and time; it seems that compression provides clear benefits.

5.6. Analysis of vertex quantization

Our scenes contain meshes which have been quantized a-priori to the minimum precision necessary for consistent visual quality. To demonstrate the effect of this a-priori quantization on compression, we quantize the *bee* mesh globally with different numbers of bits per vertex coordinate, determine the maximum quantization error and run our algorithm to quantize each meshlet locally, such that all results stay below the maximum global quantization error. The results of this experiment with 8–16 bits per vertex coordinate can be found in Figure 8.

It is interesting to see that the difference in vertex precision between our local quantization and a global quantization is not constant for matching decoding errors. There are two main reasons for that: (1) corners are quantized in the global coordinate frame, *i.e.* their quantization bits usually match the global quantization, causing outliers in the average vertex size. (2) The decoding error does not decrease linearly with increasing quantization. Nevertheless, all results stay significantly below the vertex size of a global quantization for a given tolerance.

Table 6: *Comparison to existing methods, grouped into Class 1 (decode on CPU), Class 2 (decode into GPU memory) and Class 3 (decode on the fly). Column 'Conn.' lists the average bits per triangle (bpt) for compressed connectivity. Column 'Final' reports estimated speed in Giga-triangles per second (Gtps) for 'Both' ('Decode' plus 'Render' times) multiplied with the 'Corr[ection] factor' derived from the relative GPU speed. For Meyer* et al. *[MSGS12], we report 'min' and 'max' performance values, since the variations in their results are significant, and our extrapolated numbers may, therefore, be unreliable. For Jakob* et al. *[JBG17], no rendering speed was reported, so we conservatively assume a time of zero. For Karis et al. [KSW21], no GPU model was disclosed, so we assume an RTX 3080 (the latest generation in 2020).*

Method	Year	Class	Conn. bpt	Hardware	Decode Gtps	Render Gtps	Both Gtps	Corr. factor	Final Gtps
Draco [Goo22]	2023	1	2.0	CPU					
Gurung [GLLR13]	2013	1	6.0	CPU					
Meyer [MSGS12] (min)	2012	2	4.6	Nvidia GTX 580	0.05	0.0034	0.00	$\sim 15\times$	$\sim$0.05
Meyer [MSGS12] (max)	2012	2	4.6	Nvidia GTX 580	1.72	0.65	0.47	$\sim 15\times$	$\sim$7.07
Jakob [JBG17]	2017	2	3.3	Nvidia GTX 980Ti	0.34	0.00	0.34	$\sim 4\times$	$\sim$1.37
Karis [KSW21] (Nanite)	2020	3	17.0	Assumed: GTX 3080			1.50	$\sim 2\times$	$\sim$3.00
Nicolaev [NFR22]	2022	3	12.0	Nvidia RTX 2070 Mobile			1.14	$\sim 3\times$	$\sim$3.42
Laced wires	2023	3	16.0	Nvidia RTX 4090			13.33	$1\times$	**13.33**

5.7. Comparison to existing methods

In this section, we present comparisons to existing methods based on the numbers reported in the original publications. To make a meaningful comparison with the performance numbers originally reported (since we do not have access to the code of these works), we estimate a speed correction factor based on public GPU benchmarks [WMMS23]. We are interested in comparing the compression rate of connectivity (measured in bits per triangle, *bpt*) and the joint time to decode *and* render (in Giga-triangles per second, *Gtps*, using basic rendering, such as Phong shading).

We group the methods into three classes. Class 1 contains methods for decoding on the CPU [Goo22, GLLR13], which serve as a reference for the state of the art in connectivity compression rate (but not speed). Class 2 are GPU methods decoding into memory [MKSS12, JBG17]. Class 3 are GPU methods decoding on the fly. In addition to our approach, we know of only two such methods [NFR22, KSW21].

Table 6 summarizes the data. We see that classes 1 and 2 can deliver connectivity compression which is more or less close to the theoretical optimum (1–2 bpt), but the need for storing decompressed data in memory limits the maximum speed. Moreover, these methods rely on the decompressed mesh to fit into memory. Class 3 requires more memory for connectivity (12–17 bpt), but runs clearly faster in comparison. Even though the extrapolated performance numbers used in this comparison must be taken with a grain of salt, since the relative speed of GPU models is a coarse estimate, the overall picture consistently suggests that laced wires outperform all previous methods in terms of speed. This observation can be attributed to the combined usage of compactness and locality in the mesh shader.

6. Limitations

Encoding time of our approach can become quite long for large meshes, from tens of minutes to hours for large scenes. The main cause for long encoding times is exhaustive searches for optimal vertex paths to construct laced wires. Another costly step involving many transformations between coordinate systems is the search for the smallest vertex encoding with a given error tolerance. Encoding times could be significantly reduced by heuristically pruning the search and by optimizing the encoder, which currently runs on a single CPU thread. However, we did not attempt a multi-threaded (or GPU) implementation yet.

Our current representation is a trade-off between size and decoding speed. Wire headers in particular have a non-negligible memory footprint that could be further optimized, for example, by using hierarchical bounding boxes. It also seems possible to use larger meshlets, although it is unclear at this point what constraints are imposed by the cache sizes afforded by the mesh shader.

Currently, laced wires are limited to encode manifold geometry, as vertices in a wire are visited exactly once and laces assume at most two triangles adjacent to an edge. Non-manifold meshes can be handled by splitting them into multiple manifold meshes in a preprocessing step, but this solution can lead to suboptimal results if too many sub-meshes must be generated.

7. Conclusion

The ever-increasing amount of geometry data in modern 3D applications poses great challenges. In this paper, we present an approach for end-to-end rendering of compressed mesh. This approach has several advantages. Assets are regularly stored in compressed format. While compression can greatly reduce the size in non-volatile memory, the CPU has to perform decompression before submitting the data to the GPU for rendering. GPU memory footprint and bandwidth are not improved by compression. Decompression on the GPU, such as applied in Nanite, can relieve the CPU from decompression duties, making it free for other tasks. However, existing GPU decompression approaches store the decompressed data in GPU memory in addition to the compressed data (with compression buffers managed internally). This solution even increases the total pressure on GPU memory.

Our approach to tackle these challenges is based on a novel decomposition of a mesh into locally coherent primitive groups, which we call laced wires. Laced wires store connectivity and geometry, avoid vertex duplication and can be compressed, stored and transmitted independently. Small groups of laced wires make up a meshlet, *i.e*. a small triangle cluster, for which efficient culling can be performed before decoding. Decompression occurs on the fly in a mesh shader, and the results are passed directly to the rasterizer.

We have demonstrated that on-the-fly decompression of geometric data on the GPU just in time for rasterization is viable on the GPU today. As the gap between memory bandwidth and computational bandwidth is constantly widening, the relative computational overhead of decompression will decrease over time. Carefully chosen hardware enhancements, such as support for quantized attributes, may drastically improve decompression throughput at a modest cost. It would also be interested to determine how laced wires can be used to accelerate ray tracing, which would likely combine a meshlet representation with a spatial acceleration data structure.

Acknowledgements

This work was funded by Christian Doppler Society and Qualcomm Technologies Inc.

References

[BCF03] Binotto A., Comba J., Freitas C.: Real-time volume rendering of time-varying data using a fragment-shader compression approach. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics, PVG 2003* (2003), pp. 69–75.

[BPZ99] Bajaj C. L., Pascucci V., Zhuang G.: Single resolution compression of arbitrary triangular meshes with properties. *Computational Geometry 14*, 1 (1999), 167–186.

[Cal01] Calver D.: *Vertex decompression in a shader*. Shader X, 2001.

[CDE*14] Cigolle Z. H., Donow S., Evangelakos D., Mara M., McGuire M., Meyer Q.: Survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques (JCGT) 3*, 2 (2014), 1–30.

[CG08] Chen L., Georganas N. D.: Region-based 3D mesh compression using an efficient neighborhood-based segmentation. *SIMULATION: Transactions of The Society for Modeling and Simulation International 84*, 5 (2008), 185–195.

[CH09] Courbet C., Hudelot C.: Random accessible hierarchical mesh compression for interactive visualization. *Computer Graphics Forum 28*, 5 (2009), 1311–1318.

[CH22] Cassie Hoef D. P.: Directstorage 1.1 now available. DirectX Developer Blog. https://devblogs.microsoft.com/directx/directstorage-1-1-now-available/ (2022). Accessed 17 Jan. 2024.

[Cho97] Chow M. M.: Optimized geometry compression for real-time rendering. In *Proceedings of the IEEE Visualization Conference* (1997), pp. 347–354.

[CK07] Chhugani J., Kumar S.: Geometry engine optimization: cache friendly compressed representation of geometry. In *Symposium on Interactive 3D Graphics* (2007), B. Gooch and P. J. Sloan (Eds.), ACM, pp. 9–16.

[CKLL09] Choe S., Kim J., Lee H., Lee S.: Random accessible mesh compression using mesh chartification. *IEEE Transactions on Visualization and Computer Graphics 15*, 1 (2009), 160–173.

[CR04] Coors V., Rossignac J.: Delphi: Geometry-based connectivity prediction in triangle mesh compression. *The Visual Computer 20*, 8-9 (2004), 507–520.

[CSN*12] Chen G., Sander P. V., Nehab D., Yang L., Hu L.: Depth-presorted triangle lists. *ACM Transactions on Graphics 31*, 6 (2012), 160:1–160:9.

[DBGP05] Diaz-Gutierrez P., Bhushan A., Gopi M., Pajarola R.: Constrained strip generation and management for efficient interactive 3D rendering. In *Computer Graphics International* (2005), B. Guo, H. Pfister and D. Samaras (Eds.), IEEE Computer Society, pp. 115–121.

[Dee95] Deering M.: Geometry compression. In *SIGGRAPH'95: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1995), pp. 13–20.

[DSW09] Dick C., Schneider J., Westermann R.: Efficient geometry compression for GPU-based decoding in realtime terrain rendering. *Computer Graphics Forum 28* (2009), 67–83.

[Epi20] Epic: A first look at unreal engine 5. Online announcement. https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5 (2020). Accessed 17 Jan. 2024.

[FM07] Fout N., Ma K.-L.: Transform coding for hardware-accelerated volume rendering. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1600–1607.

[GLLR13] Gurung T., Luffel M., Lindstrom P., Rossignac J.: Zipper: A compact connectivity data structure for triangle meshes. *Computer-Aided Design 45*, 2 (2013), 262–269.

[GLRL11] Gurung T., Lindstrom P., Rossignac J., Luffel M.: Lr: Compact connectivity representation for triangle meshes. *ACM Transactions on Graphics 30* (July 2011), 1–8. https://www.sciencedirect.com/science/article/abs/pii/S001044851200214X

[Goo22] Google: Draco 3D data compression. https://google.github.io/draco/ (2022). Accessed 4 Dec. 2022.

[GS98] Gumhold S., Straßer W.: Real time compression of triangle mesh connectivity. In *Proceedings SIGGRAPH* (1998), pp. 133–140.

[GWH01] Garland M., Willmott A., Heckbert P. S.: Hierarchical face clustering on polygonal surfaces. In *Symposium on Interactive 3D graphics* (2001), ACM Press.

[HA15] Haar U., Aaltonen S.: GPU-driven rendering pipelines. In *SIGGRAPH 2015: Advances in Real-Time Rendering in Games* (2015).

[Hoe22] Hoef C.: Directstorage 1.1 coming soon. DirectX Developer Blog. https://devblogs.microsoft.com/directx/directstorage-1-1-coming-soon/ (2022). Accessed 17 Jan. 2024.

[Hop99] Hoppe H.: Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH'99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1999), pp. 269–276.

[HS16] Han S., Sander P. V.: Triangle reordering for reduced overdraw in animated scenes. In *Symposium on Interactive 3D Graphics and Games* (2016), C. Wyman and C. Yuksel (Eds.), ACM, pp. 23–27.

[HV01] Hao X., Varshney A.: Variable-precision rendering. In *Symposium on Interactive 3D Graphics* (2001), pp. 149–158.

[IA02] Isenburg M., Alliez P.: Compressing polygon mesh geometry with parallelogram prediction. In *IEEE Visualization Conference* (2002), pp. 141–146.

[IL05] Isenburg M., Lindstrom P.: Streaming meshes. In *Proceedings of the 16th IEEE Visualization Conference, IEEE Vis 2005* (Minneapolis, MN, USA, Oct. 2005), IEEE Computer Society, pp. 231–238.

[Ise00] Isenburg M.: Triangle strip compression. In *Proceedings of Graphics Interface 2000* (May 2000), pp. 197–204.

[JBG17] Jakob J., Buchenau C., Guthe M.: A parallel approach to compression and decompression of triangle meshes using the GPU. *Computer Graphics Forum 36*, 5 (2017), 71–80.

[KMGL99] Kumar S., Manocha D., Garrett W., Lin M.: Hierarchical back-face computation. *Computers & Graphics 23*, 5 (1999), 681–692.

[KPRW05] Kälberer F., Polthier K., Reitebuch U., Wardetzky M.: Freelence—coding with free valences. *Computer Graphics Forum 24*, 3 (2005), 469–478.

[KSW21] Karis B., Stubbe R., Wihlidal G.: Nanite—a deep dive. Tutorials notes, SIGGRAPH 2021 Course on Advanced Real-Time Rendering. https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf (2021). Accessed 17 Jan. 2024.

[Kub18] Kubisch C.: Introduction to turing mesh shaders. SIGGRAPH Talks. https://devblogs.nvidia.com/introduction-turing-mesh-shaders/ (2018). Accessed 17 Jan. 2024.

[KXW*18] Kwan K. C., Xu X., Wan L., Wong T. T., Pang W. M.: Packing vertex data into hardware-decompressible textures. *IEEE Transactions on Visualization and Computer Graphics 24* (May 2018), 1705–1716.

[LC10] Lindstrom P., Cohen J. D.: On-the-fly decompression and rendering of multiresolution terrain. In *I3D'10: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2010), Association for Computing Machinery, pp. 65–73.

[LH04] Losasso F., Hoppe H.: Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 769–776.

[LH07] Lefebvre S., Hoppe H.: Compressed random-access trees for spatially coherent data. In *Rendering Techniques*. J. Kautz and S. Pattanaik (Eds.). The Eurographics Association (2007).

[LY06] Lin G., Yu T. P.: An improved vertex caching scheme for 3D mesh rendering. *IEEE Transactions on Visualization and Computer Graphics 12*, 4 (July 2006), 640–648.

[MKSS12] Meyer Q., Keinert B., Süßner G., Stamminger M.: Data-parallel decompression of triangle mesh topology. *Computer Graphics Forum 31*, 8 (2012), 2541–2553.

[MLDH15] Maglo A., Lavoué G., Dupont F., Hudelot C.: 3D mesh compression: Survey, comparisons, and emerging trends. *ACM Computing Surveys 47*, 3 (Feb. 2015), 1–41.

[MSGS11] Meyer Q., Sussner G., Greiner G., Stamminger M.: Adaptive level-of-precision for GPU-rendering. In *Vision, Modeling, and Visualization* (2011), P. Eisert, J. Hornegger and K. Polthier (Eds.), The Eurographics Association.

[NFR22] Nikolaev A. V., Frolov V. A., Ryzhova I. G.: 3D model compression with support of parallel processing on the GPU. *Programming and Computer Software 48* (2022), 181–189.

[PBCK05] Purnomo B., Bilodeau J., Cohen J. D., Kumar S.: Hardware-compatible vertex compression using quantization and simplification. In *HWWS'05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (New York, NY, USA, 2005), Association for Computing Machinery, pp. 53–61.

[Ros99] Rossignac J.: Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 5* (1999), 47–61.

[SKR01] Szymczak A., King D., Rossignac J.: An edgebreaker-based efficient compression scheme for regular meshes. *Computational Geometry 20*, 1-2 (2001), 53–68.

[SNB07] Sander P. V., Nehab D., Barczak J.: Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics 26*, 3 (2007), 89.

[SW03] Schneider J., Westermann R.: Compression domain volume rendering. In *IEEE Visualization, VIS 2003* (2003), pp. 293–300.

[Tau98] Taubin G.: Geometric compression through topological surgery. *ACM Transactions on Graphics 17* (1998), 84–115.

[TG98] Touma C., Gotsman C.: Triangle mesh compression. In *Graphics Interface* (1998), pp. 26–34.

[TMJ98] Tanner C. C., Migdal C. J., Jones M. T.: The clipmap: A virtual mipmap. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1998), Association for Computing Machinery, pp. 151–158.

[Ura22] Uralsky Y.: Accelerating load times for directx games and apps with gdeflate for directstorage. DirectX Developer Blog. https://developer.nvidia.com/blog/accelerating-load-times-for-directx-games-and-apps-with-gdeflate-for-directstorage/ (2022). Accessed 17 Jan. 2024.

[VSSP12] Vo H. T., Silva C. T., Scheidegger L. F., Pascucci V.: Simple and efficient mesh layout with space-filling curves. *Journal of Graphics Tools 16*, 1 (2012), 25–39.

[Wih16] Wihlidal G.: Optimizing the graphics pipeline with compute. In *Game Developers Conference* (2016).

[WMMS23] Wilson A., Miller A., Matthews M., Stevens S.: 2023 GPU benchmark and graphics card comparison chart (2023). https://www.gpucheck.com/gpu-benchmark-graphics-card-comparison-chart. Accessed 17 Jan. 2024.

[Yeu20] Yeung A.: Directstorage is coming to PC. Microsoft Developer Blog. https://devblogs.microsoft.com/directx/directstorage-is-coming-to-pc/ (2020). Accessed 17 Jan. 2024.

[YL06] Yoon S., Lindstrom P.: Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1213–1220.

[YL07] Yoon S., Lindstrom P.: Random-accessible compressed triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1536–1543.

[YLPM05] Yoon S.-E., Lindstrom P., Pascucci V., Manocha D.: Cache-oblivious mesh layouts. In *Proceedings SIGGRAPH* (2005), pp. 886–893.

Supporting Information

Additional supporting information may be found online in the Supporting Information section at the end of the article.

Supporting Information