

# 3D Model Compression with Support of Parallel Processing on the GPU

A. V. Nikolaev<sup>a,\*</sup> (ORCID: 0000-0002-7194-2471), V. A. Frolov<sup>a,b,\*\*</sup> (ORCID: 0000-0001-8829-9884),  
and I. G. Ryzhova<sup>b,\*\*\*</sup> (ORCID: 0000-0003-1613-3038)

<sup>a</sup> Moscow State University, Moscow, 1199991 Russia

<sup>b</sup> Keldysh Institute of Applied Mathematics, Russian Academy of Sciences,  
Miusskaya pl. 4, Moscow, 125047 Russia

\*e-mail: anton.nikolaev@graphics.cs.msu.ru

\*\*e-mail: vfrolov@graphics.cs.msu.ru

\*\*\*e-mail: ryzhova@gin.keldysh.ru

Received December 14, 2021; revised January 11, 2022; accepted January 16, 2022

**Abstract**—In this paper, we propose a method for 3D model compression with fast GPU-accelerated decompression. This method applies on-the-fly decompression directly in the process of rasterization or ray tracing and allows us to fit three to eight times more geometry into the same volume of GPU memory. For rasterization, we implement two variants of decompression: via geometry and via mesh shaders. For ray tracing, we use hardware acceleration with the Vulkan VK\_KHR\_ray\_tracing\_pipeline extension and propose a BVH tree leaf caching technique, which speeds up rendering nearly twofold. The conclusions made in the process of performance evaluation of the proposed method can be useful to design hardware support for 3D model compression on future GPUs, because we ran into some hardware limitations of existing GPUs.

DOI: 10.1134/S0361768822030082

## 1. INTRODUCTION

The growing detalization of 3D models used for rendering leads to the increase in the required amount of memory. In offline rendering, there are cases where scene data cannot be fully uploaded to GPU memory. In real-time rendering, available memory is also limited: it is used not only for geometry but also for scene textures, buffers, and textures that contain intermediate rendering results. Existing compression methods allow one to decompress models only before uploading them to VRAM, which reduces external memory usage but does not reduce GPU memory consumption.

## 2. EXISTING METHODS FOR 3D MODEL COMPRESSION

**Vertex data compression.** Quantization of vertex data (vertex attributes) is a widely used approach. It carries out lossy compression based on the assumption that the standard IEEE-754 floating point representation is redundant for most computer graphics applications. In [1], the range of values from minimum to maximum was divided into  $2^n$  levels, and each value was rounded to one of the two nearest levels. This approach requires storing only the minimum and

maximum values in the IEEE-754 representation and  $n$  bits of data for each compressed value.

Most of the existing methods allow one to select quantization bit counts individually for vertex positions, normals, and texture coordinates. Deltas can also be quantized to achieve the same precision using fewer bits. Values to compute deltas are chosen depending on a compression algorithm. It is also possible to further compress the result by using entropy coding [2].

**Connectivity data compression.** Generalized triangle strips were one of the first approaches to connectivity data compression. This approach combines the triangle strip and triangle fan traversals and allows one to add new triangles that do not share any vertex indices with the previous triangles. In this method, one of three cases (restart, replace oldest, and replace middle) is associated with each index to specify the reuse of vertices from previous triangles [2].

There is a large group of methods that implement step-by-step surface compression by sequential addition of triangles while storing the traversal history. Cut-Border [3] is an example of these methods. This algorithm divides a 3D model into two—inner and outer—parts, which contain processed and non-processed triangles, respectively. On each iteration, one of the triangles from the interface between these parts is

selected and one of six cases for it is determined. Each case has its unique code; for some cases, an additional parameter is also specified. The Edgebreaker algorithm [4] belongs to the same group. It is based on triangle traversal during which one of five symbols (C, L, R, E, or S) is recorded for each triangle. At each step, for a selected boundary edge, an untraversed triangle containing this edge is processed. The symbol is chosen depending on the location of the vertex of the triangle opposite to the processed edge at the boundary of the compressed region. Angle Analyzer [5] implements some ideas of Edgebreaker and Cut-Border: it also uses five symbols (as Edgebreaker) to describe different cases. For instance, it introduces case J (when two boundaries are connected) to store an additional offset from the current vertex. This method can be used for 3D models that contain both triangle faces and quad faces, and it also provides slightly better compression rate.

Valence-driven approaches are also popular. In these algorithms, the traversal also begins with a triangle; then, the boundary of the processed part is expanded, and valences of vertices are stored. One of the first implementations of this approach was proposed in [6]. In that method, all vertices of the first triangle are added to a list. Next, a vertex is selected from the list, its outgoing edges are traversed, new vertices are added to the list, and the valence of the current vertex is written. Then, valences can be processed with arithmetic coding to achieve higher compression rate. In certain cases, this algorithm requires special symbols or additional vertices for its correct operation. In [7], this approach was modified to improve compression efficiency by using a more optimal traversal of the vertices in the list, which makes it possible to avoid the frequent use of vertex addition and special characters.

There are also progressive compression methods. They use step-by-step detailing change by applying a certain operation on a base model, which is much simpler than the original one [8]. There are approaches that use step-by-step merging of vertices [9] that share the same edge (with edge splitting in the process of decompression), approaches based on vertex removal [10], etc. In [11], an approach that combines the compression by gradual detailing change and the procedural generation of models was proposed. In [12], a spatio-temporal segmentation method for compression of animated 3D models was developed. In [13], another approach to compress animated model sequences under condition of constant connectivity data was described. It was proposed to construct simplified models, which are then used to obtain the required element of the sequence. In [14, 15], existing compression methods were considered and some general directions, which correspond to those described above, were determined.

In summary, all these approaches share the same flaw (despite different ideas implemented): the impos-

sibility of large-scale parallelism in the process of decompression because of the data dependency between each previous and subsequent steps of decompression. This implies that all these methods are not quite suitable for direct decompression on GPUs in the process of rendering, because the speedup provided by the GPU is mostly due to highly parallel computations.

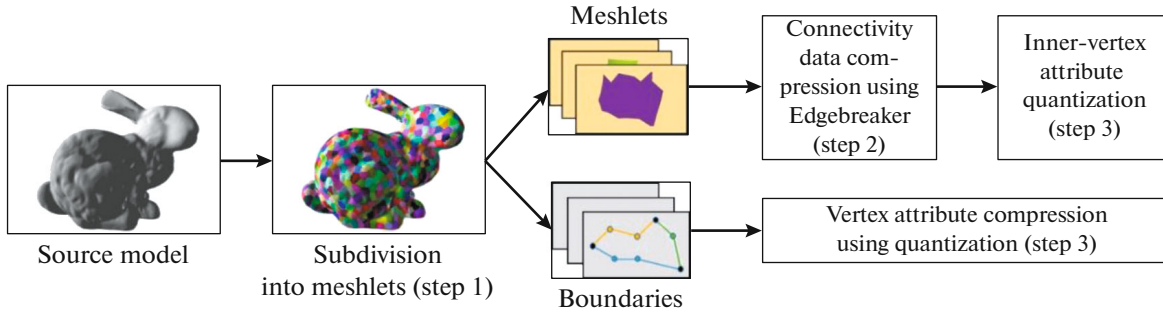
**BVH tree compression and geometry compression for ray tracing.** In ray tracing, there is a separate direction: BVH tree compression. It is used to speed up intersection search because, in massive scenes, even the BVH tree itself can require a significant amount of memory, while its compact representation speeds up ray tracing due to better cache usage [16, 17]. In [18], a large number of existing methods for BVH tree compression and geometry compression in ray tracing applications were described. In addition, we can mention several approaches, like, for example, hierarchical quantization (where vertex attribute quantization depends on the bounding volume of BVH tree leaves) [19], compressed BVH with random access using delta coding for indices [20], transformation of indexed geometry into triangle strips, approaches with zero memory cost for the tree [21, 22], etc. [18].

The main flaw of these studies is that they investigate compression only for ray tracing applications and do not consider geometry compression for efficient rasterization. In our study, this required a significant modification of base algorithms. In addition, the use of mesh compression in combination with hardware-accelerated ray tracing on GPUs has not yet been widely investigated.

### 3. PROPOSED METHOD

The main idea of the proposed algorithm is the subdivision of a source 3D model into parts (meshlets) with subsequent individual compression of each part. Meshlets are groups of triangles such that, from each triangle in a meshlet, another triangle can be reached by traversal only between the group triangles that have common edges. To improve the efficiency of compression, groups of vertices shared by two meshlets (hereinafter, they are referred to as boundaries) are stored separately, which makes it possible to avoid data duplication. The flowchart of the proposed compression method is shown in Fig. 1.

Delta quantization is used for boundary compression and compression of inner meshlet vertices. Connectivity data of each meshlet are compressed using the Edgebreaker algorithm [4]. A similar approach was proposed in [23]; however, its authors used meshlets with a significantly larger number of triangles, because their main goal was to decompress only the part of a 3D model that is visible from the current position of the camera. That allowed them to reduce resource usage in the process of decompression. In addition,



**Fig. 1.** Proposed compression method. Connectivity compression is separated from attribute compression, which uses quantization from the meshlet bounding box.

the method [23] used Angle Analyzer for connectivity data. In this work, we use short (less than 64) sequences of triangles and focus on the possibility of parallel decompression of a 3D model compressed with the described method.

Our approach enables individual processing of each meshlet, and the decompression of all meshlets can be carried out in parallel. In the proposed approach, the decompression results are not stored in memory but are directly passed to graphics (or ray tracing) pipeline, so the model data stored in the GPU memory are the same as those in the external memory. This is important for steaming optimization, when the model can be uploaded to the GPU memory directly from the hard drive in background [24, 25].

**Meshlet generation.** The subdivision algorithm used in this work is similar to the one proposed in [23]. It is based on the repeating execution of two iterations: the selection of meshlet center triangles and the addition of other triangles to meshlets until the meshlets from the previous iteration differ from the newly computed ones. Centers of meshlets are uniformly distributed across the model before the first iteration. To add triangles to meshlets, weight function (1) is used. This function allows us to obtain regular-shaped meshlets, which improves the ratio of the boundary vertex count to the inner vertex count. In addition, this function is used to create meshlets that have small differences between normals of triangles and also have approximately the same number of vertices (which does not exceed a certain threshold).

$$W_{t,m} = \left( \frac{F_m}{F_a} + C(F_m) \right) P(c_t, c_m) (\lambda - \langle n_t, n_m \rangle), \quad (1)$$

$$C(N) = \begin{cases} 0, & N < N_{\max} \\ C_m, & N \geq N_{\max}, \end{cases} \quad (2)$$

where  $F_m$  is the number of triangles in the current meshlet,  $F_a$  is the average number of triangles per meshlet,  $c_t$  is the center of a triangle,  $c_m$  is the center of a meshlet,  $n_t$  is the triangle normal,  $n_m$  is the meshlet normal,  $\lambda$  is the parameter that characterizes the

importance of planarity and compactness,  $N_{\max}$  is the maximum number of triangles per meshlet, and  $C_m$  is a constant that determines the contribution of the constraint on the number of triangles per meshlet.

The main difference from the algorithm proposed in [23] is the explicit constraint on the maximum number of triangles per meshlet. It is introduced because geometry shaders have certain limitations: they require specifying the maximum number of generated triangles and impose a constraint on the total weight of generated vertex attributes. Table 1 compares the proposed weight function with the original one. However, even with the modified function, the number of triangles in some meshlets still does not satisfy the constraint, which is why these meshlets are divided into two parts until the constraint is met.

**Meshlet connectivity data compression.** As mentioned above, the Edgebreaker algorithm is used for meshlet compression. The meshlet boundary is considered already compressed; as the initial edge, the first edge of the first boundary in the meshlet boundary list is selected. The frequencies of the symbols (C, L, R, E, and S) are also analyzed. Since the shape of meshlets for any model depends mostly on the weight function used for subdivision into meshlets, the distribution of the symbols can be assumed constant. This allows us to fix symbol codes. On test models, the following distributions were obtained: 34–42% for R, 24–29% for C, 12–17% for E, 9–14% for S, and 5–12% for L. Using Huffman coding, 2-bit codes for C, R, and E and 3-bit codes for L and S are generated. In some rare cases, these codes may prove not optimal. However, the history data occupy less than 5% of the total size of a compressed model (see Fig. 3), and even in the worst case, memory usage is not significant (less than 2.5% of the overall model size). For this reason, we do not use arithmetic coding.

**Vertex attribute compression.** Vertex positions are compressed by quantization. Absolute values are used for the first and last vertices of each boundary, while the other vertices use delta quantization. To reduce memory usage for leading zeroes upon quantization (in the binary representation), which occur because

**Table 1.** Comparison between the subdivision made using the weight functions from [23] and the subdivision made using the proposed function on the Stanford Dragon model

Function	The number of meshlets with the number of triangles from the specified range			
	[0, 32)	[32, 64)	[64, 72)	>72
Proposed one	1276	18474	216	23
Version from [23]	1285	18435	232	38

deltas are compressed with the same number of bits as absolute values, extra bits are added to each final value per component. These bits specify one of two formats for writing position of vector components: the quantization with full bit count  $n$  or the format storing only  $m$  trailing bits, where  $m$  is an integer from  $[1, n - 1]$ , which is selected by iterating over all values to minimize the size of the final model. Value  $m$  is the same for all components of all vertex positions. If the optimal value is not found, then it is set to  $n$  and the extra bit is not added. Figure 2 shows an example for  $n = 8$  and  $m = 4$ .

To compress normals, normal vectors are translated to spherical coordinates, which makes it possible to store two values instead of three, because the radius always has the unit length. Delta compression is not used.

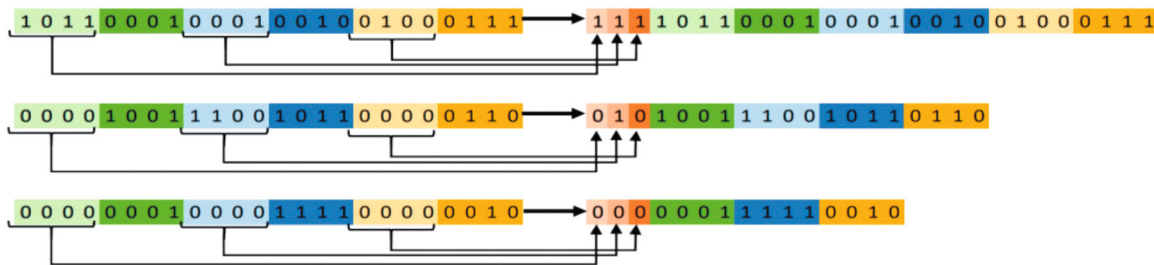
To compress texture coordinate, we use quantization of absolute values. In this case, delta compression with the coordinate representation described above can also be employed. However, texture coordinates require fewer bits as compared to vertex positions (for the compression parameters used, each texture coordinate is represented as two 12-bit values at most, as compared to three values for each position with the same or larger number of bits). Meanwhile, this approach provides additional compression for them by no more than a factor of 1.5 (on test models) while increasing decompression complexity. That is why this approach is not used.

#### 4. DECOMPRESSION ON THE GPU

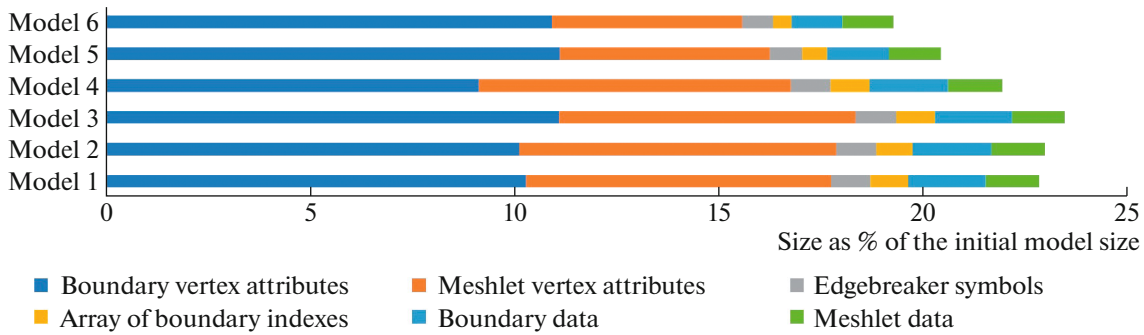
**Rasterization using geometry shaders.** For decompression in the process of rasterization, geometry shaders are used as a convenient tool available on modern GPUs. Each geometry shader thread successively decompresses triangles of one meshlet. The main disadvantage of this approach is the low performance of geometry shaders, which is due to hardware limitations. In most hardware GPU implementations, triangles from the geometry shader cannot be sent to rasterization one by one. First, a predefined number of triangles should be written to L1 cache (in our case, it is the maximum number of triangles per meshlet, because all of them depend on each other); then, these triangles are sent to rasterization. The size of L1 cache is limited; with a large number of triangles in a meshlet, geometry shaders work with low GPU occupancy (the GPU cannot run enough threads), which causes a slowdown [26].

In addition, the projection of vertices onto the screen space from the model space is usually carried out through multiplying vertex coordinates by one or several  $4 \times 4$  matrices. In the case of model decompression by means of the proposed algorithm, this operation is executed after the complete decompression of a meshlet in the geometry shader and can be carried out only by iterating over all vertices, which reduces the performance.

However, despite the problems described above, in the classic rasterization pipeline, there is no other stage for decompressing the meshlets compressed by the proposed method. Vertex and tessellation shaders cannot emit more than one vertex in each invocation,



**Fig. 2.** Vertex position compression on three samples, using eight bits for quantization and four bits to store a part of a quantized value. Components of the position vector upon quantization are shown on the left-hand side. Low-order bits are dark-colored, while high-order bits are bright-colored. On the right-hand side, the first three bits are controlling bits of each component. They are followed by blocks of four low-order and four high-order bits (they are stored only if at least one bit in the block is not zero).



**Fig. 3.** Model data upon compression (as a percentage of the initial model size) when using 64 triangles per meshlet. It can be seen that the compressed meshlet attributes (orange rectangles), together with the rest of the compressed data, take up almost the same amount of memory as the uncompressed attributes of the boundary vertices (with respect to which the quantization is performed). Therefore, for 64 triangles in a meshlet, the proposed approach is well balanced. To further improve the compression efficiency, it is required to increase the meshlet size, which reduces the decompression parallelism.

while each subsequent vertex needs data from the previous one. In addition, tessellation not only imposes a constraint on the number of vertices, it also limits mutual arrangement of triangles in a meshlet. This makes it difficult to develop an algorithm to find a subdivision that meets these requirements for an arbitrary model.

**Rasterization using geometry shaders.** A partial solution to the above problems of decompression with geometry shaders is the Vulkan API's standard extension called mesh shaders. This extension was presented by NVIDIA; it is supported by the NVIDIA Turing architecture and subsequent generations of NVIDIA GPUs [27]. It modifies the classical graphics pipeline by replacing all its stages before the rasterizer with two programmed stages: task shaders and mesh shaders.

In the proposed algorithm, the task shader carries out a partial restoration of triangle vertex indices based on the Edgebreaker traversal history. For symbol C, which specifies the addition of a new vertex, it also decompresses quantized deltas and specifies the indices of the vertices from which these deltas are computed. Then, these indices and vertex descriptions (position deltas, other attributes, and indices of previous vertices) are passed to the mesh shader.

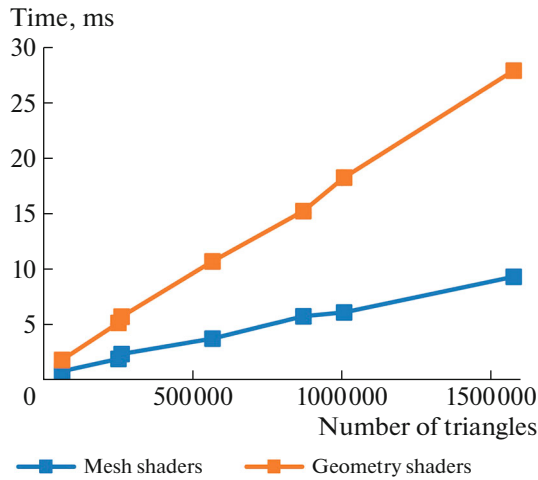
The mesh shader runs with 16 threads per meshlet. It decompresses boundary vertices in parallel (vertices of each boundary can be processed only sequentially because of delta compression; however, boundaries do not depend on each other and are processed in separate threads). Then, in one of the threads, global positions of inner vertices are generated based on the already available absolute values at boundaries and the deltas with vertex indices that are received from the task shader. This operation is relatively simple (it involves a small number of simple vector additions), which is why it does not significantly reduce the efficiency of parallel mesh shader execution. Upon computing the absolute values, the parallel multiplication

by matrices is carried out in 16 threads. Finally, the indices from the task shader and the generated coordinates are written as the results of mesh shader execution.

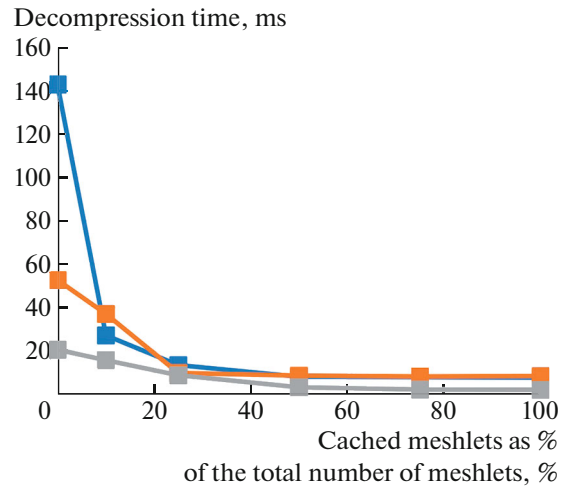
**Decompression in the process of ray tracing.** For decompression in the process of ray tracing, we use the Vulkan standard extension—VK\_KHR\_ray\_tracing\_pipeline—which adds support of acceleration structures and introduces a new pipeline used for ray tracing.

To find intersections with a compressed 3D model, intersection shaders are used. As an accelerating structure, a set of axis-aligned bounding boxes (AABBs) is used. Coordinates of AABB nodes are computed at the application startup by using the compute shader, which carries out a one-time decompression of the 3D model and finds the minimum and maximum coordinates of each meshlet along each coordinate axis. In the intersection shaders, the index of a meshlet is determined based on the index of the bounding box the intersection with which is currently checked. To check the intersection between a ray and a meshlet, decompression is first carried out; then, the intersection of the ray with each triangle of the meshlet is checked. If there are several intersections, then the nearest one is selected. However, this implementation has a very low performance. As compared to rasterization (where each primitive and meshlet is processed only once per frame), in ray tracing, the number of rays is the main factor. Each ray requires the decompression of the meshlet the intersection with which is checked, which can be done only in one thread (in VK\_KHR\_ray\_tracing\_pipeline, several threads cannot be launched in the intersection shader). This problem is partially solved by caching the decompressed meshlets (see below).

**Caching for ray tracing acceleration.** The main idea of this optimization is based on a preliminary decompression of a fixed number of meshlets by using compute shaders. By varying the number of cached mesh-



**Fig. 4.** Comparison of decompression performance with geometry shaders and mesh shaders (carried out on NVIDIA RTX 2070 Mobile).



**Fig. 5.** Comparison of decompression performance depending on the number of meshlets in the cache. Carried out for 921 600 traced rays. Different colors are used for different models.

lets, we can find a balance between memory usage and ray tracing performance.

Meshlets to be cached are selected by counting the number of intersection shader launches for each meshlet. The atomicAdd operation is used to increment the counter of intersections with the current meshlet by one for each intersection check. Since the direct obtention of these data for the current frame requires extra computations, values from the previous frame are used while assuming small changes between the frames. Based on these values, the meshlets not present in the cache are decompressed (they are written instead of the meshlets that are no longer in the cache). Then, the intersection shader checks the presence of the meshlet in the cache, and either the intersection with the already decompressed meshlet is checked or the decompression is preliminarily carried out.

Figure 5 shows the tracing performance versus the cache size.

### 5. DISADVANTAGES OF THE PROPOSED ALGORITHM

Most of the limitations of the proposed method are due to the subdivision into meshlets. This algorithm requires smoothing the majority (at least) of the normals of a 3D model; otherwise, the subdivision operation fails to find adjacent triangles, creating a lot of small meshlets and reducing the efficiency of compression. Another disadvantage of the meshlet subdivision algorithm is its performance on models with small sets of triangles that do not have common vertices with the rest of the model. This reduces the efficiency of compression: the description of the meshlet does not change, whereas the number of triangles in it

decreases, causing an increase in the number of bits per triangle.

### 6. ANALYSIS OF THE RESULTS

**Compression efficiency evaluation.** To evaluate the efficiency of compression, it is reasonable to compute the compression rate for each resulting component.

For vertex data, the quantization makes it possible to reduce the amount of the data required for each vertex from 256 bits (eight IEEE-754 floating-point values) to 76 bits (when using the following quantization parameters: 12 bits per vertex position, 8 bits per normal, and 12 bits per texture coordinate), while the use of the reduced bit count for the quantization of position deltas provides an additional reduction in the amount of these data by a factor of 1.23 to 1.34 (on test 3D models). This allows us to achieve the total size of about 69 bits (the total amount of vertex data is reduced by more than 3.7 times). In practice, storing

**Table 2.** Analysis of the decompression performance limitations in rasterization on the Blender Suzanne and Stanford Dragon models. (Based on the NVIDIA Nsight Graphics data [28].) The decompression time differs from that shown in the results because, when measuring in Nsight, GPU boost was disabled

Model	Shader	Warps that cannot launch due to lack of L1 cache	Decompression time
Suzanne	Geom.	50.8%	1.25 ms
Suzanne	Mesh	14.2%	0.54 ms
Dragon	Geom.	77.5%	19.1 ms
Dragon	Mesh	28%	7.13 ms



Fig. 6. 3D models used for comparison.

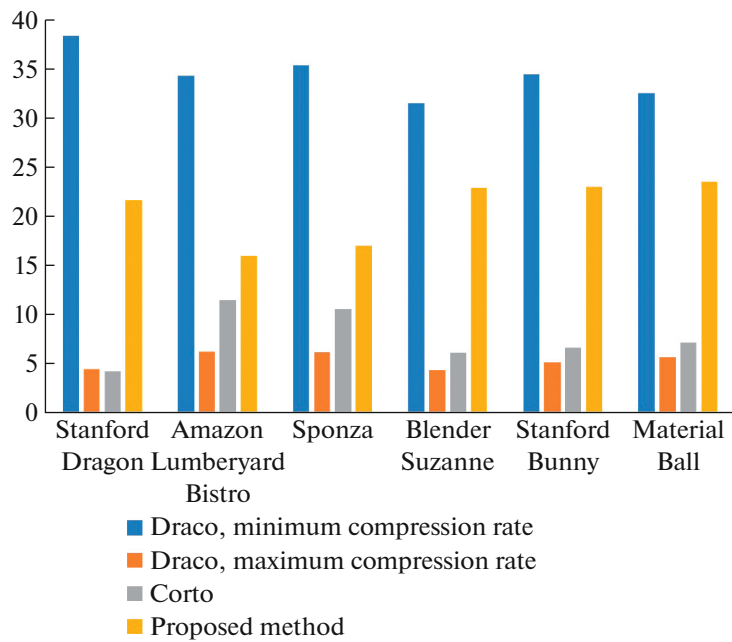


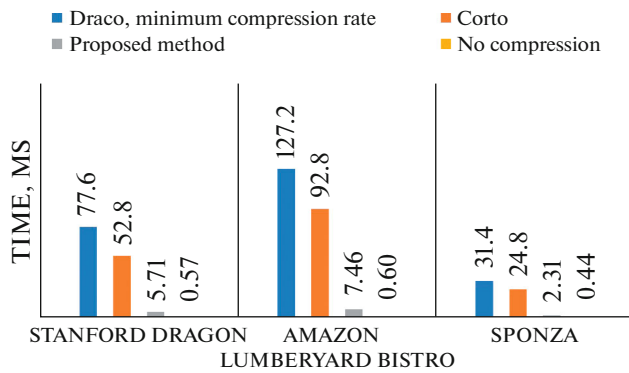
Fig. 7. Size of the 3D model as a percentage of its initial size.

both the start and end points of the boundary slightly reduces the compression efficiency (e.g., by a factor of 3.7 to 3.1 on test models).

For connectivity data, the Edgebreaker algorithm makes it possible to use less than 3 bits (2.19–2.22 on test models) per triangle, instead of 32 bits per each of three vertices, thus enabling the compression by a factor of 32 and higher. The amount of additional data depends on the number of meshlets and the number of boundaries, which exceeds the number of meshlets by an average factor of 2.9 (with each meshlet using an average of 5.5 boundary indices) on test 3D models.

This results in approximately 400 bits of additional data per meshlet. If we attribute the additional data to the connectivity data, then the final result is approximately 11.63 bits per triangle on test models, which is equivalent to an 8.25-fold compression.

Thus, depending on the volume of vertex data and connectivity data in a 3D model, it is possible to achieve a compression rate from 3.1 to 8.25. On the test models where connectivity data occupies about 40% of the initial volume, the compression makes it possible to reduce the amount of the required memory by a factor of 4.26 to 4.35. It can be seen from Fig. 4



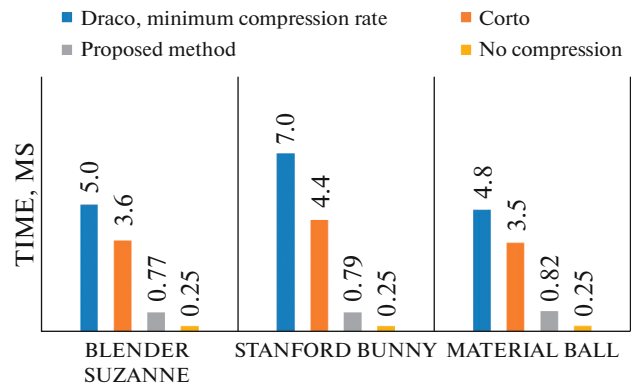
**Fig. 8.** Comparison of decompression times, carried out on RTX 2070 Mobile for the proposed method and the version without compression, as well as on Intel Core I7 10750H for the other methods as they do not support parallel decompression on the GPU.

that vertex attributes constitute the major part of data in the compressed model, and a further reduction in their size is hardly feasible. This requires either the reduction in the number of quantization bits (this can be done in some tasks; however, in the general case, it reduces the accuracy and introduces visual distortions) or the use of entropy coding for these data, which significantly complicates the decompression and reduces the performance.

**Decompression performance analysis.** Figures 4 and 5 show the decompression performance when using geometry shaders and mesh shaders and the decompression performance for ray tracing depending on the cache size. It is found that the main factors that limit the performance is the amount of the data written by geometry shaders and mesh shaders, as well as the size of the arrays used for temporary data storage. This effect is most noticeable when using geometry shaders. In fact, less than 15% of the available GPU computing resources are used. The corresponding results are shown in Table 2. In the case of decompression for ray tracing, it can be seen that, starting from 50% of the cached meshlets, the performance remains almost the same. This is due to the fact that some of the meshlets are almost always covered by other meshlets. The bottleneck here is the check of intersections with all meshlet triangles for each ray, which is implemented in the intersection shader without any hardware acceleration.

**Comparison with existing methods.** For comparison with the existing methods, we used the decompression with mesh shaders on two open-source libraries: Draco and Corto. Both libraries support data quantization and delta compression, as well as use (in one form or another) triangle traversal for connectivity data.

The comparison was carried out on six 3D models (see Fig. 6): Blender Suzanne (62976 triangles and 32057 vertices), Stanford Bunny (33528 vertices and



**Fig. 9.** Comparison of decompression times (continuation).

65630 triangles), Material Ball (31310 vertices and 61088 triangles), Sponza (184330 vertices and 262267 triangles), Stanford Dragon (438929 vertices and 871306 triangles), and Amazon Lumberyard Bistro (1020907 vertices and 762263 triangles; only the indoor scene was used).

For quantization, the following values were used in all the cases: 12 bits per vertex positions, 8 bits per normal, and 12 bits per texture coordinates. The tests were carried out on NVIDIA RTX 2070 Mobile and Intel Core I7 10750H. For the performance comparison of the methods on the CPU, the average decompression time for 5 runs was computed; the time required for uploading the 3D model to RAM was not taken into account. The results of the comparison on six models with different characteristics are shown in Figs. 7–9. The comparison in terms of the memory required to store these models on the GPU is not shown because, for proposed method, it is the same as in Fig. 7, whereas the other methods store decompressed models.

## 7. CONCLUSIONS, DISCUSSION, AND FUTURE RESEARCH

The proposed approach increases the amount of geometry that can be placed in the GPU memory by a factor of 3 to 8. Moreover, it is almost always possible to achieve a fourfold increase without visible geometric distortions, even when the camera approaches the surface. However, the potential slowdown of visualization is quite significant (from 5 to 10 times). In real-world applications, it can be less due to the fact that the pipeline is performance-limited at other stages (e.g., in the fragment shader). The performance analysis using special tools [28] has shown that the existing GPU pipelines have the following limitations that become bottlenecks. (1) For rasterization using geometry shaders, it is necessary to store all decompressed vertex data in the L1 cache of the GPU (shared memory) before sending the entire meshlet for rasterization. (2) For rasterization using mesh shaders, it is



required to write data when transferring them from the task shader to the mesh shader, as well as use L1 cache to store temporary data for decompression, because not all of them can be placed in registers. (3) For ray tracing, the data decompression at the leaves limits the performance.

## REFERENCES

1. Chou, P.H. and Meng, T.H., Vertex data compression through vector quantization, *IEEE Trans. Visualization Comput. Graphics*, 2002, vol. 8, no. 4, pp. 373–382.
2. Deering, M., Geometry compression, *Proc. 22nd Annu. Conf. Computer Graphics and Interactive Techniques*, 1995, pp. 13–20.
3. Gumhold, S., Improved cut-border machine for triangle mesh compression, *Proc. Erlangen Workshop*, 1999, vol. 99, pp. 261–268.
4. Rossignac, J., Edgebreaker: Connectivity compression for triangle meshes, *IEEE Trans. Visualization Comput. Graphics*, 1999, vol. 5, no. 1, pp. 47–61.
5. Lee, H., Alliez, P., and Desbrun, M., Angle-analyzer: A triangle-quad mesh codec, *Comput. Graphics Forum*, 2002, vol. 21, no. 3, pp. 383–392.
6. Touma, C. and Gotsman, C., Triangle mesh compression, *Proc. Graphics Interface*, 1998, pp. 26–34.
7. Alliez, P. and Desbrun, M., Valence-driven connectivity encoding for 3D meshes, *Comput. Graphics Forum*, 2001, vol. 20, no. 3, pp. 480–489.
8. Abderrahim, Z., Techini, E., and Bouhleb, M.S., State of the art: Compression of 3D meshes, *Int. J. Comput. Trends Technol.*, 2012, vol. 4, no. 6, pp. 765–770.
9. Hoppe, H., Progressive meshes, *Proc. 23rd Annu. Conf. Computer Graphics and Interactive Techniques*, 1996, pp. 99–108.
10. Cohen-Or, D., Levin, D., and Remez, O., Progressive compression of arbitrary triangular meshes, *IEEE Visualization*, 1999, vol. 99, pp. 67–72.
11. Uyttensprot, S., Mesh compression and procedural content generation, 2018.
12. Luo, G. et al., Spatio-temporal segmentation based adaptive compression of dynamic mesh sequences, *ACM Trans. Multimedia Comput., Commun., Appl.*, 2020, vol. 16, no. 1, pp. 1–24.
13. Hajizadeh, M. and Ebrahimnezhad, H., Eigenspace compression: Dynamic 3D mesh compression by restoring fine geometry to deformed coarse models, *Multimedia Tools Appl.*, 2018, vol. 77, no. 15, pp. 19347–19375.
14. Maglo, A. et al., 3D mesh compression: Survey, comparisons, and emerging trends, *ACM Comput. Surv.*, 2015, vol. 47, no. 3, pp. 1–41.
15. Elmas, A.A., Investigation of single-rate triangular 3D mesh compression algorithms.
16. Mahovsky, J.A., Ray tracing with reduced-precision bounding volume hierarchies, 2005.
17. Ylitie, H., Karras, T., and Laine, S., Efficient incoherent ray traversal on GPUs through compressed wide BVHs, *Proc. High Performance Graphics*, 2017, pp. 1–13.
18. Meister, D. et al., A survey on bounding volume hierarchies for ray tracing, *Comput. Graphics Forum*, 2021, vol. 40, no. 2, pp. 683–712.
19. Segovia, B. and Ernst, M., Memory efficient ray tracing with hierarchical mesh quantization, *Proc. Graphics Interface*, 2010, pp. 153–160.
20. Kim, T.J. et al., RACBVHs: Random-accessible compressed bounding volume hierarchies, *IEEE Trans. Visualization Comput. Graphics*, 2009, vol. 16, no. 2, pp. 273–286.
21. Eisemann, I.M., Bauszat, P., and Magnor, M., Implicit object space partitioning: The no-memory BVH, Technical Report, Computer Graphics Lab, 2011, vol. 365.
22. Chitalu, F.M., Dubach, C., and Komura, T., Binary ostensibly-implicit trees for fast collision detection, *Comput. Graphics Forum*, 2020, vol. 39, no. 2, pp. 509–521.
23. Choe, S. et al., Random accessible mesh compression using mesh chartification, *IEEE Trans. Visualization Comput. Graphics*, 2008, vol. 15, no. 1, pp. 160–173.
24. AMD Smart Access Memory. <https://www.amd.com/en/technologies/smart-access-memory>. Accessed June 3, 2021.
25. Thompson, A. and Newburn, C., GPUDirect storage: A direct path between storage and GPU memory, NVIDIA Developer Whitepapers, 2019, vol. 8.
26. Barczak, J., Why geometry shaders are slow, 2015. <http://www.joshbarczak.com/blog/?pf7>. Accessed May 5, 2020.
27. Kubisch, C., Introduction to turing mesh shaders, 2018. <https://developer.nvidia.com/blog/introduction-turing-mesh-shaders>. Accessed May 5, 2020.
28. NVIDIA Nsight Graphics. <https://developer.nvidia.com/nsight-graphics>. Accessed June 3, 2021.

Translated by Yu. Kornienko