Mesh compression method with on-the-fly decompression during rasterization and streaming support

Anton Nikolaev
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Moscow, 119991 , Russia
anton.nikolaev@graphics.cs.msu.ru

Alexandr Shcherbakov
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Moscow, 119991 , Russia
alex.shcherbakov@graphics.cs.msu.ru

Vladimir Frolov
Keldysh Institute of Applied Mathematics RAS
Miusskaya sq., 4
Moscow, 125047, Russia
Lomonosov Moscow State University
vfrolov@graphics.cs.msu.ru

ABSTRACT

In this article we propose a method of mesh compression and streaming, that can be used for real-time rendering applications. While most of other existing compression methods require decompression on CPU before rendering and streaming methods use only non-compressed models, our approach allows to reduce memory consumption by applying decompression on GPU during rendering and streaming of only needed geometry data at the same time. Proposed approach requires pre-processing step, on which coarse 3D model with quad faces is build and resampling is done. Afterwards, each face of model is compressed and can be later rendered with tessellation shaders (decompression is done during rasterization). Also, we propose a way of adding streaming support to our compression method to further reduce memory consumption. Finally, we made a comparison with state of the art approach levels of detail (LODs) approach and found that proposed approach has much lower memory consumption without negative effects to rasterization performance and quality.

Keywords
Meshes, compression, streaming, parallel decompression.

1 INTRODUCTION

3D model compression is often necessary. With growth of mesh quality the amount of memory required to store these meshes increases too. Also, a GPU with better computation performance can be required to support real time rendering when using higher quality meshes. For now several solutions of this problem exist.

2 EXISTING SOLUTIONS

The first solution of the described problem is mesh compression. Almost all compression approaches use vertex data quantization (lossy compression technique, which compresses a range of values to a single quantum value) [Cho02] and try to reduce as much as possible the size of connectivity data or don't store it at all. First steps in this direction were done by applying triangle strips and triangle fans approaches, which allow storing only one index of new vertex per triangle instead of storing all three indices. Also some generalizations can be used [Dee95]. Triangle traversal approaches make another large group of mesh compression methods. Such approaches allow replacing indices with traversal history data. Several symbol codes can be used, specifying placement of new triangle vertices on the border of the compressed area [Gum99] [Ros99]. Sometimes additional data is also required to be stored with the history. Several such approaches also allow combining both quad and triangle faces in the same mesh [Lee02]. Another group of mesh compression techniques is formed by valence-driven methods. Vertex valences are written during traversal instead of traversal history, but sometimes additional data is also needed [All01]. Progressive compression is also possible and it is based on applying of some simplification operation (vertex or edge removal, etc) to the source mesh until the coarse mesh will be generated. During decompression, opposite operations are applied in the reversed order to the coarse mesh to restore the original one [Hop96] [Coh99]. However, all described approaches share one common problem. Parallel decompression on GPU is impossible due to data dependency between iterations of the decompression process. So the decompressed model would be stored in VRAM and GPU memory usage won't change. Mesh compression method, allowing random access to mesh parts was also proposed [Cho08]. It is achieved by splitting a source mesh into parts (meshlets). By decreasing meshlet size parallel decompression can be applied [Zha12]. The authors used mesh segmentation and afterwards each part was compressed independently from all others. Decompression was done once on GPU with Cuda. Methods with parallel GPU decompression were also proposed [Jak17] [Mah21], however they use Cuda or compute shaders to decompress the mesh once before usage. It increases decompression speed comparing to CPU implementation, however GPU memory usage is still the same. Several compression methods were proposed for GPU decoding during rendering. However, they mostly apply only quantization for vertex attributes and possibly do some initial mesh simplification [Cho97] [Cal02] [Hao01]. Finally, there exist compression method suitable for multi-resolution meshes that uses hierarchy data to improve compression rate, but parallel decompression on GPU also is not supported [Käl09].

Levels of detail can reduce memory consumption allowing only the needed part of the model to be loaded into GPU memory [Lue03]. For complex scenes including a huge number of separate objects the HLODs approach [Eri01] also exist. It proposes to build LOD not only for separate objects, but also for groups allowing to increase LOD quality without negative memory usage impact. LODs generation in run-time was also proposed to be used for dynamic scenes.

Mesh streaming (loading of only necessary mesh data instead of loading the whole mesh) can also be used, but most existing solutions propose only mesh transfer over the network [Kim04]. Partial load of mesh data from external storage to RAM for more effective processing (including compression) is also described [Ise05], but in all cases only the decompressed mesh is used during rendering [Dou19].

In conclusion, the problem of mesh compression method not requiring decompression before rendering is still open. The aim of our work was to develop such method. In addition we decided to use some version of streaming or LODs with our proposed compression algorithm to further decrease memory consumption.

3 PROPOSED APPROACH

The proposed approach is based on generating a coarse mesh with quad faces (later named patches). Afterwards, the resampling (generation of an almost regular mesh using the original one and the coarse one) is done to achieve the required quality (chosen by the user). Finally compression is applied to the obtained data. With certain limitations on the resampling process the resulting data can be rendered with tessellation shaders, which are available and hardware accelerated on most of modern GPUs.

Coarse model generation and resampling

Coarse meshes are generated using the open-source tool Instant Meshes [Jak15]. This step requires certain user involvement to control several parameters, which cannot be computed automatically.

Afterwards the source mesh is resampled using GPU accelerated ray-tracing. For this process we use *rayQuery* from tessellation shaders. The acceleration structure is built from the source model. For each point in a uniformly tessellated patch a ray is traced along the interpolated normal to find the intersection point on the source mesh. The position of intersection becomes the position of this point. Finally this result is saved to buffer and moved to the GPU when all user-controlled setup is finished.

Such an approach achieves performance acceptable for interactive parameter tuning during the resampling process. The user can control mesh quality (controlled by the tessellation factor) and the maximum distance between source point and ray intersection point and see the resulting mesh in real-time with the ability to move the camera. Actually this resampling method can be replaced by some more complex approach. The only requirement is the same vertex and polygon placement pattern, that is used by the tessellation. Vertex positions can vary and are not required to be uniformly placed over the patch.

Compression

To compress all vertex data quantization is used. Bit count is selected separately for patch corners, inner vertices and patch borders depending on required precision (also specified by the user). For border vertices and inner vertices not absolute values but deltas (differences of values from some predicted ones) were quantized. Also, it should be noted, that quantization bit counts are selected for the whole mesh, not per patch or per border.

Deltas for borders are computed from points, appearing during subdivision of edges into equal parts (number of parts depends on number of border vertices). Transformation to new coordinate system, made by edge direction and two orthogonal axis, is also done for edge deltas. For inner vertices deltas are computed from positions appearing in the tessellation with uniform vertex placement. Coordinate system transformation is applied to inner vertex deltas too and uses interpolated normal and patch edge direction to form orthonormal basis. (See fig. 1)

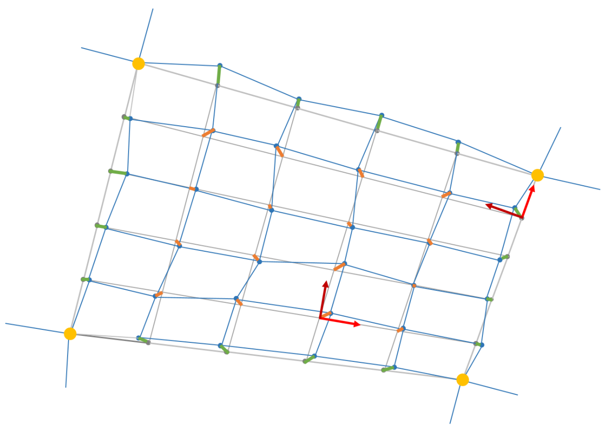

Figure 1: Visualization of saved data on one patch (projected along normal for simplicity). Corner vertices are shown yellow color, resampled grid is shown with blue lines and points. Deltas for border vertices are shown by green lines, inner vertex deltas are shown by orange lines. Grey grid shows a uniform subdivision of the patch. Axes for one border vertex and one inner vertex are shown with red arrows.

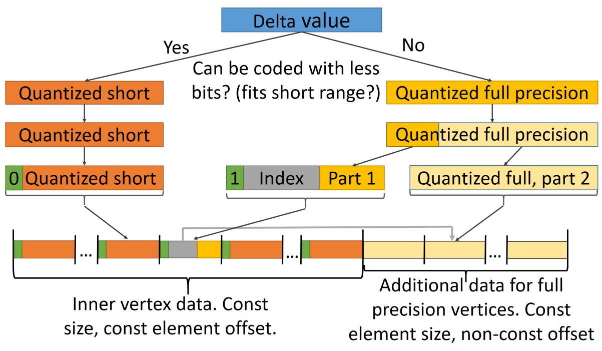
Delta value
Yes
No
Can be coded with less bits? (fits short range?)
Quantized short
Quantized full precision
Quantized short
Quantized full precision
0 Quantized short
1 Index
Part 1
Quantized full, part 2
...
Inner vertex data. Const size, const element offset.
Additional data for full precision vertices. Const element size, non-const offset

Figure 2: Visualization of proposed coding scheme.

For many patches inner vertex deltas are small especially along axes close to patch edges. To save some memory amount on heading zeroes appearing during small deltas quantization with constant bit count we propose a coding scheme with less number of bits for such cases. Supposing that m bits will be used for quantization following data representation is used. $(m+1)$ bits are stored for each inner vertex. One leading bit specifies value format. The following m bits contain either quantized delta with low bit count (if source delta value was small enough) or index of additional data and part of quantized value (if all indices in all patches fit less than m bits). For each patch additional data is placed after the main data. It allows saving some memory on heading zeroes for small deltas and still supports random access to data of each coordinate, so no decoding of previous vertices is required. Bit counts for both quantization formats are common for all patches of the mesh. The proposed coding scheme is shown on fig. 2. To obtain the final value one memory read from the precomputed address and possibly one more read operation from the address computed using the previously read value (required only for vertices quantized with full bit count) are required. The number of bits for short deltas is computed by iterating through all possible values (the optimal range is also found for each case) to use as small amount of memory as it is possible. As indices of additional data don't use much memory, while a lot of quantized deltas have heading zeroes, the proposed approach allows to save more than 10% of memory used by inner vertices data (see fig. 3).

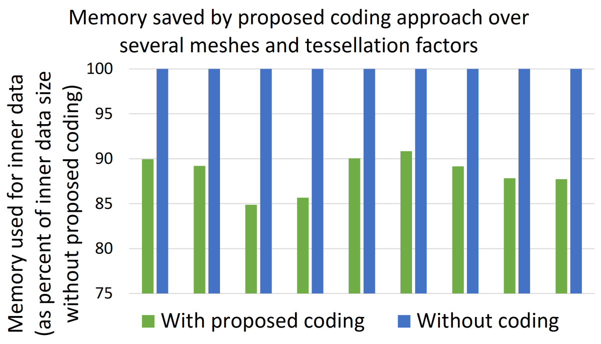

Figure 3: Memory saved by proposed coding scheme.

Per patch data is also needed and it stores corner vertex and border indices (because this data is shared between several patches) and offset of inner data in buffer, containing inner data (this value can't be simply computed during decompression, because each patch can have own inner data size).

Decompression

Decompression is done on the GPU during rasterization and uses tessellation shaders. Data is passed in 4 storage buffers (corner vertex data, border data, inner data and patch descriptions), each of them is presented as unsigned 32-bit integer array. The tessellation control shader decodes some common per patch data like corner vertex indices and positions and sets the tessellation factor. The tessellation evaluation shader restores either one inner vertex position or border vertex position. The total cost of one vertex decode includes a read of per-patch parameters. Each packed value with not-power-of-two bit count requires one or two memory reads (each value can either fully be placed in a 32-bit unsigned integer or be split between two subsequent integers and require two reads). So decoding cost also includes 1-2 memory reads for border vertex or 1-4 reads for inner vertex (one or two packed values) per each of 3 position components. Finally several MAD and bit-wise operations are required per component.

Streaming

As stated before, we store only the compressed model in GPU memory, but with streaming memory consumption can be improved even more, by storing only needed parts of mesh. However, due to the specifics of the proposed compression approach (inner and border data is stored separately, quantized inner vertex deltas have their own packed format) streaming required some changes.

Each patch can use its own level of detail independent from other patches. We just introduce a limitation for total amount of memory used by all mesh data. Actually it is a configurable parameter and can be set depending on the application. The only data that should always be placed into GPU memory is patch descriptions, including border addresses, level of inner tessellation, and indices of corner vertices. Also corner vertices should always be placed in GPU memory. Actually part of this data is used to support streaming, while other part can be thought about as a minimal presented LOD of the mesh. During rendering the tessellation control shader can write requests for inner vertex data load into a request buffer if edge size gets greater than a defined value (it is also application controlled). In the next frame the CPU reads back these values and produces a buffer of processing requests for the compute shader. Also border tessellation factors are computed during this process. The compute shader decompresses inner vertex data and border data, does the interpolation of positions depending on the required tessellation factors and packs all this data with the described compression algorithm. However, due to data packing the inner data size can't be predicted before interpolation. So we use additional temporary buffer for processed data, where each patch has enough space to be placed not depending on interpolation results. Sizes of obtained data are also written to this buffer. Later we read this data on the CPU side and select where to copy these results. Such an approach doesn't noticeably increase memory usage as the size of all these buffers is constant not depending on mesh size and is also configurable. During streaming patch description data is also updated. Addresses of borders are selected on the CPU before compute shader dispatch and are written by this shader. For the inner data addresses are generated based on sizes from temporary buffer and written to patch descriptions during copying. Tessellation factors in patch descriptions (for inner factor) and in border data are also updated.

Memory management and defragmentation

The main problem of the proposed approach comes from data fragmentation. During camera movement some patches of the mesh can require better quality, so their tessellation factors should be increased and new data should be loaded. In most of cases new inner or border data can't be loaded in place of previous data, as its size increases, so a new place for it is found in buffer. And even after space used by previous patch data is marked as free it still can't be fully utilised, because with the proposed coding scheme its hardly possible to find several blocks with exactly matching size. So after several such iterations more and more memory is wasted, until there will be no space for required data in the buffer.

Use of standard solutions such as paging is hardly possible. Inner vertex position data (and border data that takes even less size) take too small an amount of memory to use this value as page size: such a choice will require a lot of additional memory to support a page table with huge number of pages. At the same time, large pages will introduce internal fragmentation. So we developed own method of memory management and defragmentation. It is supported on the CPU side and tries to fit as much data as possible into fixed size buffers. If there is not enough space to load required data the defragmentation process happens.

For defragmentation a temporary fixed-size buffer is used. The sequence of free and used blocks starting and ending with used blocks is searched with the requirement to maximize the size of free blocks inside. Another restriction is total size of used blocks in sequence, as all these blocks should fit the temporary buffer. When the sequence is found, all used blocks are shifted to the end of last used block placed before the sequence, producing one free block made from several ones met in sequence. If such sequence can't be found, the maximum sequence of used blocks placed between two free blocks and fitting into the fixed-size buffer is searched for and merged with another used sequence. Both operations are done by first copying all used blocks into the temporary buffer with the compute shader to avoid numerous buffer copy operations. Afterwards the filled contents of the temporary buffer are copied back by one operation. Only one such operation is applied each frame to reduce the performance impact. Such an approach has a fixed memory (and actually can have even zero memory cost as some currently unused buffer can be used as temporary too) and performance cost, but also controls the maximum amount of memory wasted due to fragmentation. It can be seen that this algorithm can't do anything only in one case: when we have a sequence of free and used blocks, where none of the free blocks is suitable for new data allocation, while each sequence of used blocks is larger than temporary buffer. The worst case appears when the buffer is filled by sequences of used blocks with a bit greater size than temporary buffer has, while free blocks between them have size a bit less than required for per patch inner vertex data. So even having temporary buffer with size of tens of patches would result in several percent of memory wasted due to fragmentation in worst case (at the same time full a mesh can include hundreds or thousands of patches, so size of temporary buffer is quite small). Finally, the worst case wouldn't generally be present for a long time since on camera movement some patches can request lower tessellation factor and will be replaced, leaving some free space.

If defragmentation fails, the memory management system tries to downgrade some blocks, that are not visible or don't require high tessellation factor any more. Afterwards defragmentation is applied again, if the block for new data still cannot be allocated.

Limitations

In its current version the proposed approach applies only to vertex positions. In general, it can be extended to compress also normals and texture coordinates. The only problem is the resampling process on texture coordinate seams.

Also, the described approach will show low compression efficiency on meshes containing a lot of separate parts each with a few triangles. This problem comes from the generation of the coarse model. Each such part will generate a patch, that will require to store a patch description for each such part.

Finally some compression steps involve user interaction, as we can't choose some parameters automatically. During coarse model generation we need to get as low patch count as possible, but still save certain mesh properties (new gaps shouldn't appear, patches should be placed to allow resampling of any visually important part of the source mesh, etc). A minimal amount of patches is desirable, because each patch always stores its description and corner vertices in GPU memory and this size is not dependant on the required tessellation factors of patches. Also, triangle and pentagon patch faces produced by instant meshes should be removed if possible. Each triangle face is presented as a quad with one fictive vertex. Pentagon faces are converted into 5 triangles and same procedure is applied afterwards. An alternative solution is to split such faces into 3 or 5 quads, but it produces T-junctions. And subdivision of all faces to remove these T-junctions is not acceptable, as it increases number of patches in the coarse model by 4 times, that will be even more inefficient.

4 EXPERIMENTAL RESULTS AND COMPARISON

To compare memory usage of proposed approach with LODs we used different tessellation factors the same across all model (actually its worst case for streaming) and compared memory consumption with an LOD, having same number of faces as the tessellated surface. From fig. 4 and fig. 5 it can be seen that with minimal level of detail our method uses a bit more memory comparing to an LOD, as it uses more memory for patch descriptions (4 indices, inner vertex data and border data addresses, tessellation factor) than LOD per face (just 4 indices). However, when the quality increases, our method shows much better memory usage, as we use delta coding with quantization, packing for internal data and don't need to store indices for border and inner vertices of each patch.

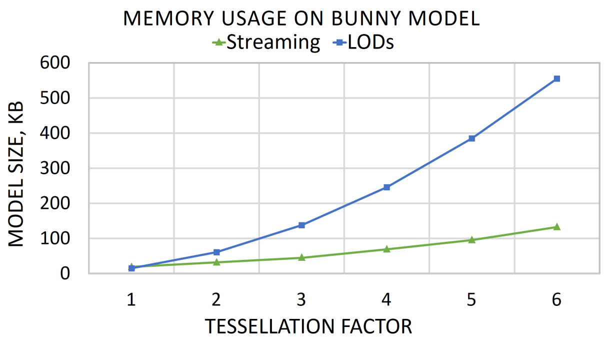

Figure 4: Comparison between the proposed compression method and an LOD with same number of polygons as the tessellated surface has (Stanford Bunny model).

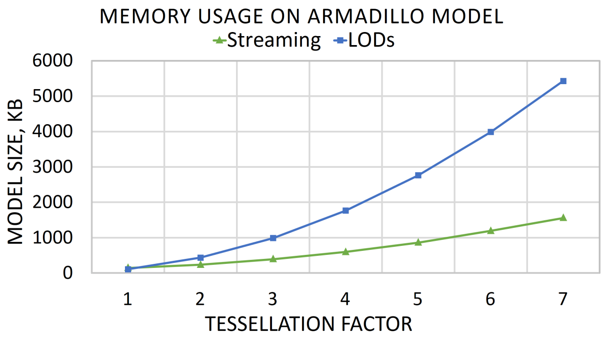

Figure 5: Comparison between the proposed compression method and an LOD with same number of polygons as the tessellated surface has (Armadillo model).

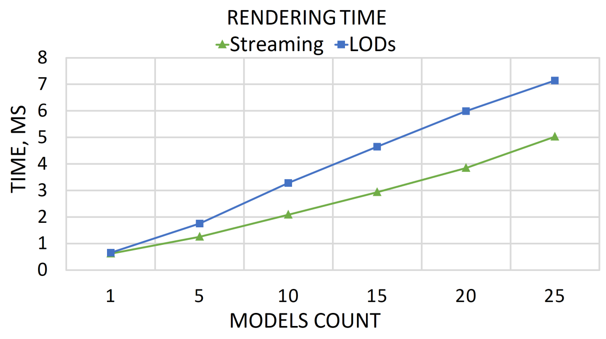

Figure 6: Comparison of rendering time with LODs and proposed approach.

Also, performance of rendering with both LODs and proposed approach was measured. We found that proposed approach can be used for real-time applications. Figure 6 shows that our method can work even faster

Edge length	Distance	LODs (kb)	Streaming (kb)
0.605	2.891	19.230469	18.90625
0.075	2.401	37.828125	18.90625
0.401	0.736	19.230469	19.105469
0.054	2.891	74.941406	19.929688
0.048	2.401	74.941406	20.839844
0.177	0.736	148.359375	21.539062
0.041	2.891	148.359375	23.367188
0.136	0.736	293.789062	24.050781
0.034	2.401	148.359375	27.671875
0.095	0.736	583.6875	29.210938
0.082	0.736	583.6875	32.234375
0.027	2.891	293.789062	35.550781
0.027	2.401	293.789062	35.5625
0.02	2.891	583.6875	52.527344
0.027	0.736	1154.882812	61.8125
0.0	0.736	1154.882812	70.742188
0.014	2.401	583.6875	81.511719
0.014	2.891	1154.882812	83.984375
0.0	2.401	1154.882812	120.609375

Table 1: Memory usage comparison with streaming and LODs depending on distance to mesh and required maximum edge length (in screen space coordinates). Zero edge length means to load all patches and LOD in maximum quality.

than LODs. In both cases the performance was actually limited by the primitive clipping and culling stage, not by memory access or SM (Streaming Multiprocessor) usage, showing that proposed approach has low computation cost and acceptable number of memory reads. Finally, some other work can be done on the GPU at the same time (for example pixel shaders or async compute queue tasks), as SM units are loaded by less than 33% according to Nvidia NSight.

Also, we compared our proposed approach with LODs on different camera positions with the Stanford Bunny model. For LODs we were supporting median will edge size not greater than a specified value. The same value was used as maximum edge length in each patch for streaming with compression. Results are shown in table 1. It can be seen that LODs use same amount of memory only when the minimal tessellation factor on all patches and minimal LOD are selected (lines 1 and 3 in table). In other cases LODs use much more memory.

Figures 7, 8, 9, 10 and 11 show visual comparison of LODs and the proposed approach with different camera positions and desired edge sizes. For all pictures in this section LODs are shown with a white mesh, while the proposed approach shows colored mesh (patches with different tessellation factors with different colors). Also, all comparisons are done with wireframe rendering mode to show polygon sizes. Regions of interest are

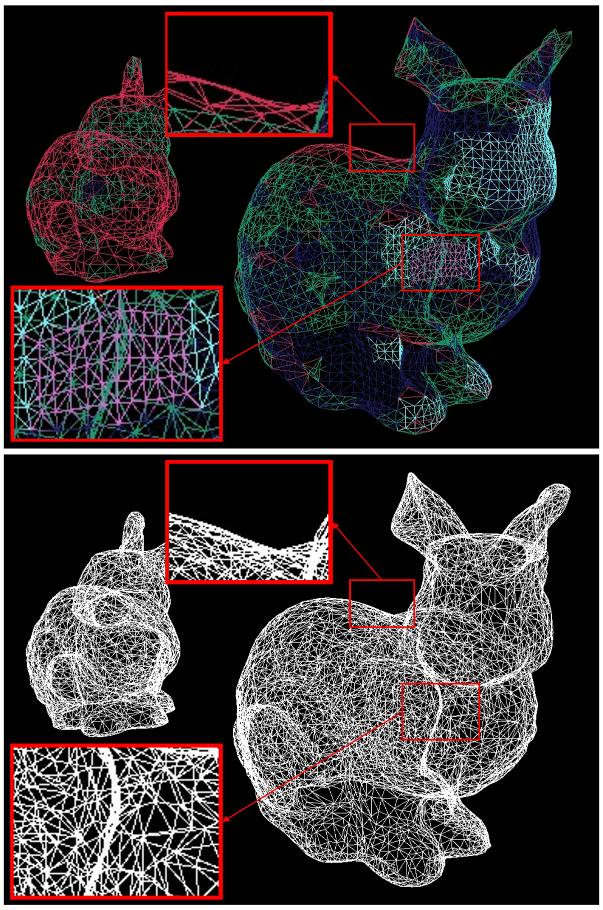
Figure 7: Visual comparison of rendering with LODs and the proposed approach.

shown with red rectangles. It can be seen that for proposed approach mesh is built in a more optimal manner. For mesh parts close to camera faces have similar size for both methods, while far parts require less polygons with the proposed approach, as they don't require such quality. Finally, fig. 12 shows sample render of Armadillo 3D model by proposed streaming and compression method.

5 CONCLUSION

In this paper we proposed a novel approach, that compresses models up to three times according to experimental results. Also, we describe a streaming and memory management systems, which can be used to further improve the memory usage. Finally, we made some comparisons, showing that our approach is fast enough and has effective VRAM usage at the same time and can be used in real-time applications. However, several limitations still exist, mostly caused by coarse mesh generation process difficulties. Also, some improvements can be done to extend the method for more general usage scenarios with compression of not only vertex positions, but also texture coordinates and normals.

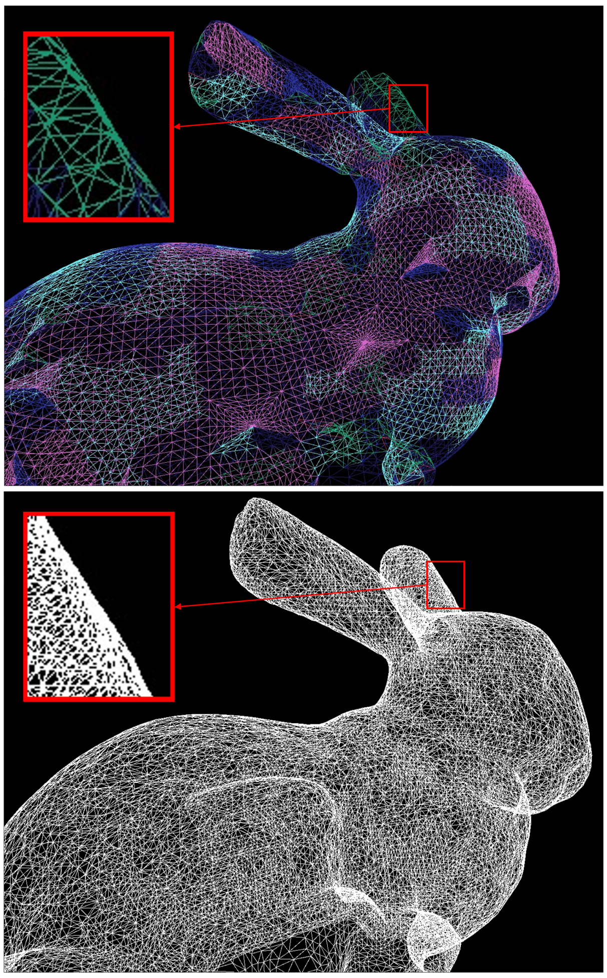
Figure 8: Visual comparison of rendering with LODs and the proposed approach.

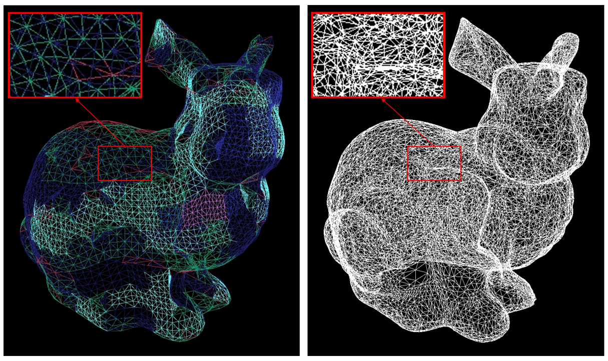
Figure 10: Visual comparison of rendering with LODs and the proposed approach.

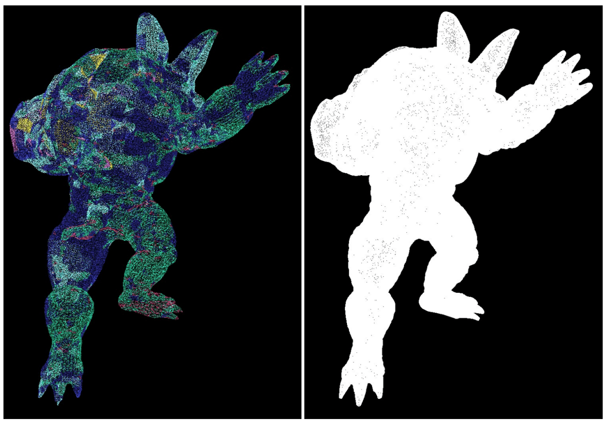
Figure 11: Visual comparison of rendering with LODs and the proposed approach. It can be seen that the proposed approach generates less number of sub-pixel triangles.

REFERENCES

[All01] Pierre Alliez and Mathieu Desbrun. "Valence-driven connectivity encoding for 3D meshes". In: *Computer graphics forum*. Vol. 20. 3. Wiley Online Library. 2001, pp. 480–489.

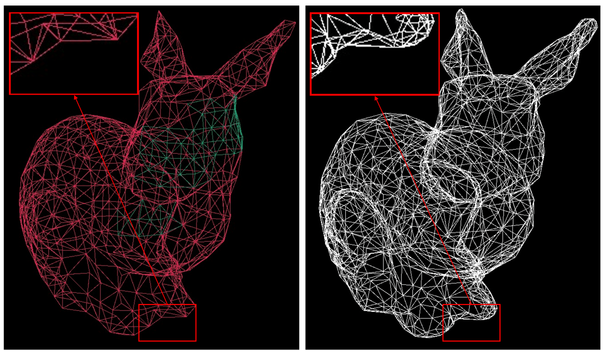
Figure 9: Visual comparison of rendering with LODs and the proposed approach.

[Cal02] Dean Calver. "Vertex decompression in a shader". In: *ShaderX: Vertex and Pixel Shader Tips and Tricks* (2002), pp. 172–187.

[Cho02] Peter H. Chou and Teresa H Meng. "Vertex data compression through vector quantization". In: *IEEE Transactions on Visualization and Computer Graphics* 8.4 (2002), pp. 373–382.

[Cho08] Sungyul Choe, Junho Kim, Haeyoung Lee, and Seungyong Lee. "Random accessible mesh compression using mesh chartification". In: *IEEE Transactions on Visualization and Computer Graphics* 15.1 (2008), pp. 160–173.

[Cho97] Mike M Chow. *Optimized geometry compression for real-time rendering*. IEEE, 1997.

[Coh99] Daniel Cohen-Or, David Levin, and Offir Remez. "Progressive compression of arbitrary triangular meshes". In: *IEEE visualization*. Vol. 99. 1999, pp. 67–72.

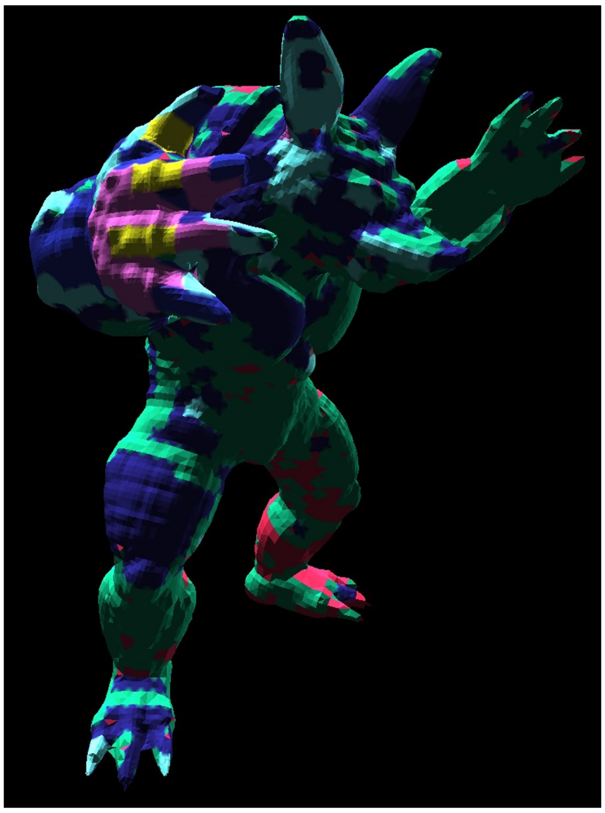

Figure 12: Sample of rendered 3D model with the proposed approach.

[Dee95] Michael Deering. “Geometry compression”. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 1995, pp. 13–20.

[Dou19] Alexandros Doumanoglou, Petros Drakoulis, Nikolaos Zioulis, Dimitrios Zarpalas, and Petros Daras. “Benchmarking open-source static 3D mesh codecs for immersive media interactive live streaming”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.1 (2019), pp. 190–203.

[Eri01] Carl Erikson, Dinesh Manocha, and William V Baxter III. “HLODs for faster display of large static and dynamic environments”. In: *Proceedings of the 2001 symposium on Interactive 3D graphics*. 2001, pp. 111–120.

[Gum99] Stefan Gumhold. “Improved cut-border machine for triangle mesh compression”. In: *Erlangen Workshop*. Vol. 99. 1999, pp. 261–268.

[Hao01] Xuejun Hao and Amitabh Varshney. “Variable-precision rendering”. In: *Proceedings of the 2001 symposium on Interactive 3D graphics*. 2001, pp. 149–158.

[Hop96] Hugues Hoppe. “Progressive meshes”. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, pp. 99–108.

[Ise05] Martin Isenburg, Peter Lindstrom, and Jack Snoeyink. “Streaming compression of triangle meshes”. In: *ACM SIGGRAPH 2005 Sketches*. 2005, 136–es.

[Jak15] Wenzel Jakob, Marco Tarini, Daniele Panozzo, and Olga Sorkine-Hornung. “Instant field-aligned meshes.” In: *ACM Trans. Graph.* 34.6 (2015), pp. 189–1.

[Jak17] Johannes Jakob, Christoph Buchenau, and Michael Guthe. “A Parallel Approach to Compression and Decompression of Triangle Meshes using the GPU”. In: *Computer Graphics Forum*. Vol. 36. 5. Wiley Online Library. 2017, pp. 71–80.

[Käl09] Felix Kälberer and Konrad Polthier. “Lossless compression of adaptive multiresolution meshes”. In: *2009 XXII Brazilian Symposium on Computer Graphics and Image Processing*. IEEE. 2009, pp. 80–87.

[Kim04] Junho Kim, Seungyong Lee, and Leif Kobbelt. “View-dependent streaming of progressive meshes”. In: *Proceedings Shape Modeling Applications, 2004.* IEEE. 2004, pp. 209–220.

[Lee02] Haeyoung Lee, Pierre Alliez, and Mathieu Desbrun. “Angle-analyzer: A triangle-quad mesh codec”. In: *Computer Graphics Forum*. Vol. 21. 3. Wiley Online Library. 2002, pp. 383–392.

[Lue03] David Luebke, Martin Reddy, Jonathan D Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.

[Mah21] Ahmed H Mahmoud, Serban D Porumbescu, and John D Owens. “RXMesh: a GPU mesh data structure”. In: *ACM Transactions on Graphics (TOG)* 40.4 (2021), pp. 1–16.

[Ros99] Jarek Rossignac. “Edgebreaker: Connectivity compression for triangle meshes”. In: *IEEE transactions on visualization and computer graphics* 5.1 (1999), pp. 47–61.

[Zha12] Jie-Yi Zhao, Min Tang, and Ruo-Feng Tong. “Connectivity-based segmentation for GPU-accelerated mesh decompression”. In: *Journal of Computer Science and Technology* 27.6 (2012), pp. 1110–1118.