Packing Vertex Data into Hardware-Decompressible Textures

Kin Chung Kwan, Xuemiao Xu, Liang Wan, Tien-Tsin Wong, and Wai-Man Pang

Abstract—Most graphics hardware features memory to store textures and vertex data for rendering. However, because of the irreversible trend of increasing complexity of scenes, rendering a scene can easily reach the limit of memory resources. Thus, vertex data are preferably compressed, with a requirement that they can be decompressed during rendering. In this paper, we present a novel method to exploit existing hardware texture compression circuits to facilitate the decompression of vertex data in graphics processing unit (GPUs). This built-in hardware allows real-time, random-order decoding of data. However, vertex data must be packed into textures, and careless packing arrangements can easily disrupt data coherence. Hence, we propose an optimization approach for the best vertex data permutation that minimizes compression error. All of these result in fast and high-quality vertex data decompression for real-time rendering. To further improve the visual quality, we introduce vertex clustering to reduce the dynamic range of data during quantization. Our experiments demonstrate the effectiveness of our method for various vertex data of 3D models during rendering with the advantages of a minimized memory footprint and high frame rate.

Index Terms—vertex data compression, real-time rendering, hardware texture compression, permutation, GPU acceleration

1 Introduction

With the increasing demand for ultra-realistic rendering of game scenes, the number of 3D objects, textures, and vertex data to be stored in the memory of a graphics processing unit (GPU) is frequently enormous. A real-time game scene often has a number of small-scale 3D objects (hundreds to thousands of polygons each, shown in Fig. 1) rendered. Increasing the number of objects presented in a game improves the richness of the game content, and increasing the mesh resolution of each object reduces unpleasant polygonal artifacts. The trade-off for such increases is obviously an increase in storage requirements. Although modern GPUs support hardware encoding of textures, storing, and direct decoding of compressed textures in GPU memory for real-time rendering, vertex data are seldom stored in GPU memory in compressed form because no hardware circuit is tailored for real-time decoding.

Although geometry compression methods [2], [3], [4] can effectively compress meshes, they are not designed for the real-time rendering demands of game applications. For real-time rendering, the mesh may be accessed in a random order (*random-accessibility*) because of the arbitrary viewing angle. Note that many existing geometry compression methods can decode vertex data only in sequential order. Moreover, if the mesh is stored in a compressed form in GPU memory and decoded only when it is displayed, a decoding speed of at least 30 fps is required for real-time purposes. Although modern GPUs are programmable (shader programming [5] or GPGPU [6]), achieving such a decoding speed is challenging because of the complexity of decoding methods.

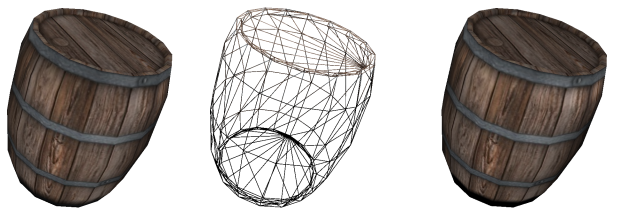

(a) Original (b) Wireframe (c) Our compressed result

Fig. 1: An example of a simple 3D model (284 polygons) in game applications. The result in (c) has a compression ratio of 1:7.

In this paper, we mainly focus on the compression of vertex data, which refers to vertex attributes, including vertex positions, normals, and texture coordinates. We propose a method to store compressed vertex data in GPU memory and allow real-time, random-order decoding of vertex data for game applications.

Our key idea is to exploit the existing hardware compression functionalities of ordinary GPUs. The basic idea is to store the vertex data in textures and utilize the texture compression hardware on the GPU to achieve real-time decoding. However, naïvely compressing the vertex data with texture compression results in unsatisfactory results (Fig. 2(c)) because of the inconsistency of the data structure. Whereas vertex data have a graph structure, texture data have a grid structure. Texture compression utilizes the data coherence among neighboring texels for effective compression. Naïvely packing the graph vertex data into texture disrupts the data coherence and reduces compression effectiveness.

To increase data coherence, we propose to permute the vertex data packed in the textures using an optimization approach. By

- *K.C. Kwan is with Department of Computer Science and Engineering, The Chinese University of Hong Kong and School of Computing and Information Sciences, Caritas Institute of Higher Education. E-Mail: kckwan@cse.cuhk.edu.hk*
- *X. Xu is the corresponding author with the South China University of Technology and The Chinese University of Hong Kong. E-Mail: xuemx@scut.edu.cn*
- *L. Wan is with School of Computer Software, Tianjin University and Department of Computer Science and Engineering, The Chinese University of Hong Kong. E-Mail: lwan@tju.edu.cn*
- *T.-T. Wong is with Department of Computer Science and Engineering, The Chinese University of Hong Kong and Shenzhen Key Laboratory of Virtual Reality and Human Interaction Technology, Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China. E-Mail: ttwong@cse.cuhk.edu.hk*
- *W.-M. Pang is with School of Computing and Information Sciences, Caritas Institute of Higher Education. E-Mail: wmpang@ieee.org*

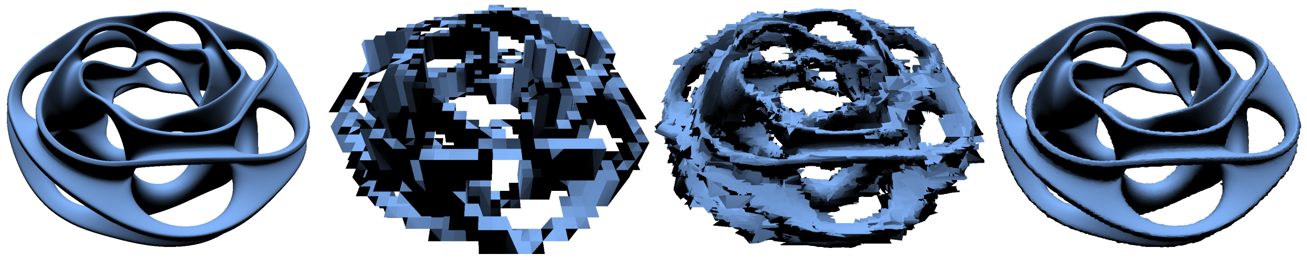

(a) Original (b) Purnomo et al. [1] (c) Without permutation (d) Our result

Fig. 2: The rendering results for a 3D model with position data compressed. All methods here use 12 bpv.

minimizing a global error metric, we obtain an optimal permutation that drastically improves the visual quality. To further improve the visual quality, we propose a clustering approach to reduce compression error. We demonstrate the effectiveness of our vertex data permutation by compressing positions, normals, and texture coordinates. In all experiments, we achieve real-time performance using the hardware texture decompression ability of modern GPUs.

2 Related Works

Geometry compression is a popular method of reducing geometry size. Deering [7] was the first to introduce the concept of geometry compression, and his algorithm sequentially encodes (decodes) the offsets between successive vertices in a vertex array. Although his decompression algorithm [7] [8] is designed for hardware implementation, his sequential decoding process cannot be parallelized on a GPU. Touma and Gotsman [9] coded a mesh using a parallelogram predictor. Isenburg [10] followed this idea and improves the compression ratio by predicting the vertex degree. Vasa and Brunnett [11] further improved the idea using a weighted parallelogram predictor. Again, their methods require sequential processing and hence cannot fit into the regular rendering pipeline.

Progressive compression methods [2] [3] [4] compress multiresolution models, but their recursive approaches are not GPU-friendly. Hao and Varshney [12] proposed to compress geometry by reducing the data precision of each graphical primitive being rendered. However, the different precisions of different vertices complicate implementation on the GPU. Gu et al. [13] remeshed 3D surfaces onto geometry images and compressed these images using wavelet-based coders. Praun and Hoppe [14] followed this idea and remeshed the 3D surface onto spheres. Rodríguez et al. [15] remeshed the object surface into tetrahedra. Their representation directly exploits an existing image or video compression technique. However, their method requires a sequential mesh reconstruction process during decompression and therefore cannot be fully parallelized on a GPU.

Spectral techniques [16] partition the mesh into manageable and local submeshes, and each submesh is then represented as a compact linear combination of orthogonal basis functions. All vertices on the same submesh or patch must be decompressed simultaneously, and thus they are not randomly accessible. However, random access is a critical requirement for our problem.

Although several existing techniques are partially accelerated using GPUs, they cannot be fully parallelized because part of the procedures must be performed sequentially. Therefore, they are inefficient if the data are decompressed purely in the GPU and still require data decompression in a CPU before transmission for rendering (Fig. 3). Although these methods achieve higher compression ratios or better quality, they are not suitable for our problem.

Schnabel et al. [17] decomposed point-cloud data into primitive shapes and represented them as height-fields. The decomposition makes it difficult to handle complex connectivity data. Meyer et al. [18] parameterized the surface normal onto an octahedron for compression. However, this method is applicable for compressing normals only. Gobbetti et al. [19] partitioned the mesh into quadrilateral patches and performed parametrization on each patch. However, their method assumes that the input is a single manifold mesh instead of multiple disconnected objects (e.g., game scenes).

Other methods such as 3D Mesh Coding (3DMC) [20] use a triangle strip compression scheme to encode connectivity. Triangle Fan-based compression (TFAN) [21] follows this idea and exploits the concept of a triangular fan. However, it mainly focuses on connectivity data, even as it simply quantizes the vertex data for compression. Grouper [22] aggregated the triangles to represent the connectivity data in a compact form. However, the vertex data are still stored in uncompressed plain form.

In contrast to previous methods, our approach does not rely on specific hardware or a complex vertex shader. Instead, we exploit a common compression technique in graphics hardware to compress vertex data. This makes our approach easy to implement, efficient, and effective for decompression on the GPU only. In other words, we can reduce the memory usage in graphics hardware to suit more or larger 3D models. Furthermore, our approach makes no assumption about connectivity; it can accommodate nonmanifold meshes or even a polygon soup. Because of the data structure of connectivity, we can randomly access the compressed data for real-time rendering (Fig. 3).

Several techniques share a similar intention as ours by attempting to store compressed data in GPU memory. Calver described the basic principles for decoding quantized vertex attributes in a vertex shader [23] [24]. Purnomo et al. [1] followed this idea and implemented a vertex data decoder on graphics hardware. Their method automatically allocates the number of bits to each vertex attribute and globally quantizes all vertex data. Lee et al. [25] proposed performing local quantization after mesh partitioning. A more comprehensive survey of mesh compression can be found in the work of Maglo et al. [26].

Other techniques related to our permutation are the mesh layout [27] or partitioning [28] for cache-efficient access. Because the neighbors of recently accessed data are prefetched, the next access is preferably to nearby data in order to access the cache. Thus, cache-efficient methods mainly focus on the coherence of the access order of data. However, the access order is not related to the texture compression error. Hence, their methods do not address the reduction in the compression error as we do.

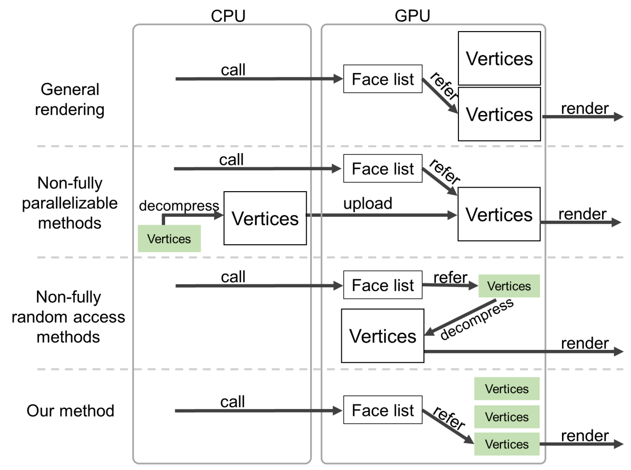


Fig. 3: The data flow for rendering using different methods. The green boxes represent compressed data.

3 Background

3.1 Rendering on Hardware

To render 3D objects with graphics hardware, vertex data must be transmitted from the CPU to the GPU. Because of the limited bandwidth between the CPU and the GPU, data are usually preloaded into GPU memory for quick access. As the vertex data alone do not define the connectivity of a 3D object, an additional array (i.e., *facelist*) is required to render the 3D objects. This facelist, which references the preloaded vertex data on the GPU, can either be stored in GPU memory or sent to the GPU in a streaming manner for rendering. This vertex-face structure is the most widely used mesh representation for real-time rendering.

Although preloading can greatly reduce the data transmission to the GPU, it demands significantly more GPU memory space because the preloaded vertex data are stored in raw and uncompressed form. Although each 3D object in a game scene may not be large in size, it is still impossible to upload all 3D objects simultaneously to the GPU memory as its capacity is limited. It is natural to consider compressing these data to regain memory space on the GPU for uploading more data. To avoid significantly affecting the rendering speed, we require that decompression be performed purely on the GPU and in a random-access manner for real-time rendering of the compressed vertex data (Fig. 3). To accommodate these requirements, our idea is to exploit the existing hardware texture compression functionality of ordinary GPUs.

3.2 Compression on Hardware

Nearly all GPUs have built-in circuits for different hardware texture compression methods, including 3Dc [29], S3 Texture Compression (S3TC) [30], and Block Partitioned Texture Compression (BPTC) [31]. Our goal is to exploit this hardware decompression to solve our problem.

We believe most block-based hardware texture compression methods should work well in our method. However, most of these methods do not have the flexibility for choosing the geometry precision or bit per vertex (bpv). Therefore, the selection of hardware texture compression can significantly affect the compression ratio of the final result. In this paper, we use 3Dc as an example for our experiments. 3Dc is a 4×4 block-based hardware texture compression using quantization. Its block structure allows 3Dc to be parallelized on the GPU hardware and randomly accessed. 3Dc stores the maximum and minimum values using 8 bits each and represents each data value in the block with 3 bits. Therefore, 3Dc can use 64 bits to represent 16 values in one block for one-dimensional data. In other words, one 3D datum (e.g., vertex position) can be represented by 12 bits (12 bpv).

Although 3Dc is a relatively old technique and is designed for normal mapping, it meets our needs for vertex data compression on the GPU. Experimental results show that our method reconstructs high-quality vertex data for rendering. Note that our method is not limited to a specific compression technique. Higher performance can be achieved if a better hardware compression technique (if it exists) is applied. However, naïvely compressing the vertex data with texture compression has unsatisfactory results. Thus, we propose a permutation approach to reduce the compression error.

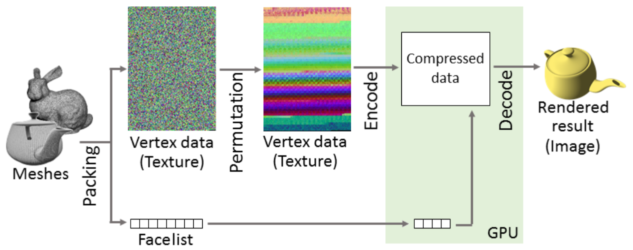


Fig. 4: The framework of our system.

4 Our Framework

Fig. 4 shows our framework. Our system mainly involves four steps: data packing, permutation, encoding, and decoding. The first three steps are performed only once to preprocess the vertex data, whereas the last step runs simultaneously with real-time rendering.

a) Data Packing: To utilize the built-in hardware texture compression, we must pack the vertex data into 2D texture. To do so, we put one vertex data into one pixel of a texture. When multiple attributes are available in vertex data (e.g., position and normal), each attribute can be stored in different textures. Therefore, vertex data are tightly packed into the texture regardless of their connectivity. Multiple disconnected meshes can also be packed into the same texture (e.g., objects in the scene in Fig. 12). To refer to the vertex data in textures, we rely on texture coordinates (u, v), which are stored in the facelist (connectivity data). In contrast to the geometric image [13], which also packs mesh into an image, our method does not require any parameterization, which may introduce additional errors.

b) Permutation: The quality of texture compression decidedly depends on data coherence in the texture. However, vertices in mesh are graph-natured, whereas pixels in texture are grid-natured. Naïvely packing vertex data into texture reduces the data coherence and generates unsatisfactory results (e.g., Fig. 2(c)). To increase data coherence in the packed vertex data, we permute the data in texture. More details are discussed in Section 5.

c) Encoding and Decoding: Once the vertex data are packed and permuted in textures, we can compress them easily by invoking the hardware API. The decompression can be performed on the GPU with a simple vertex shader during rendering. We first upload the necessary elements to the vertex shader. In the shader, we directly access the corresponding data in a compressed texture via a texture lookup function. This function automatically triggers the GPU hardware texture decompression mechanism and decompresses the data in a random-access manner instantaneously. This results in an immediate and data-on-demand decompression of vertex data at the time of rendering. This process does not require a huge memory footprint for caching intermediate decompressed data.

5 Permutation

Naïvely packing vertex data in textures introduces a number of compression errors (e.g., Fig. 2(c)) because of the lack of data coherence. To reduce such errors, we perform optimization by permuting the vertex data in the texture. This idea is analogous to cache-efficient methods [32] [33]. However, such methods rely on a static access order of the vertex for permutation. In our case, the data may be randomly accessed in a dynamic order in different frames because in real-time applications, the scenes are dynamic. Thus, cache-efficient methods cannot be directly applied to our problem.

The following sections define an error metric to evaluate a given arrangement of the vertex data to guide the optimizer to seek the best permutation for vertex data.

5.1 Attribute-Space Error Metric

In general, errors for vertex compression can be measured in two different spaces: image space [1] [34] or attribute space [35] [36]. An image space metric compares the rendering results of the altered model and the original model with multiple camera angles and lighting conditions, which can be too computationally expensive for optimization processing. In our system, we use an attribute-space metric for our metric.

Our compression involves vertex data only. It does not affect the connectivity (i.e., facelist). Therefore, we simply evaluate errors based on the root mean square error (RMSE) between the original vertex and the decompressed vertex data. Therefore, our metric to measure compression error is defined as

$$e_i(p) = \sum_{j=1}^{M} \| \mathbf{q}_{ij}(p) - \hat{\mathbf{q}}_{ij}(p) \|^2, \forall\, i \in \{1, \ldots, N\} \quad (1)$$

where p denotes the current arrangement of the vertex data textures and the indices i and j represent the i-th block and j-th dimension, respectively. M and N are the total number of dimensions and blocks of texture, respectively. $\mathbf{q}_{ij}(p)$ is the value of the attribute of the original vertex data. $\hat{\mathbf{q}}_{ij}(p)$ represents the counterpart in the reconstructed data. The operator $\| \cdot \|$ computes the 2-norm. Therefore, the average compression error $E(p)$ is calculated as

$$E(p) = \frac{1}{NBM} \sqrt{\sum_{i=1}^{N} e_i(p)} \quad (2)$$

where B is the block size of the employed hardware texture compression technique. In our experiment, B is 16 for 3Dc. This normalization process minimizes the effect of the model scale on the parameter setting.

Note that our metric only considers the move of the vertex after compression. Continuity of data is not considered. Therefore, when handling texture coordinates, our method may fail to represent the texture seam in the proximity of the continuous surface because of the compression error. One way to suppress such an error is to employ weighting on each vertex in Eq. 1 to reduce the compression error of the vertices near the seams.

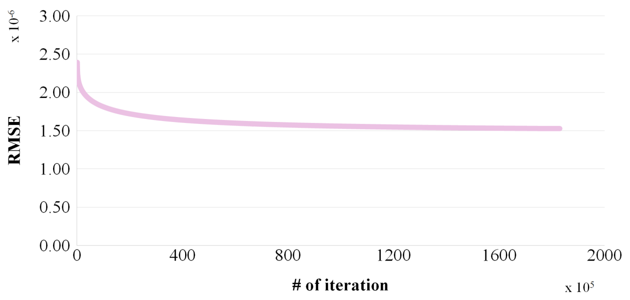


Fig. 5: Compression error (RMSE) of Fig. 2 decreases during SA optimization iterations and finally converges.

5.2 Iterative Optimization

Based on our attribute-space error metric (Eq. 2), our objective function can be formulated as the following equation:

$$p^* = \arg\min_p \{E(p)\}. \quad (3)$$

Its goal is to determine an optimal permutation p^* for minimizing the compression error. Obviously, this is a discrete optimization problem in the permutation space. Solvers for optimization problems are usually designed for continuous space and hence are not applicable to our problem. Smith et al. [37] attempted to transform permutation space problems into continuous space with sorting operations. Unfortunately, sorting in each optimization iteration is very time consuming. In our work, we exploit the simulated annealing (SA) algorithm [38] to solve this discrete optimization problem because it can easily be adapted to the permutation space. Moreover, as SA is GPU friendly, it can be significantly accelerated using the GPU. Details about GPU implementation for SA are discussed in the next section.

During every iteration of our optimization, given a texture of vertex data with an arrangement p, we randomly swapped 2 pixels in the texture to produce a new arrangement p'. Note that these 2 pixels must belong to two different blocks; otherwise, the error remains unchanged. Then, we calculated our metric to decide whether we should accept or reject this new arrangement. Obviously, if the compression error is reduced using p', we can use p' for the next iteration. Even if the compression error is slightly increased using p', we can still accept p' with an iterative-varying acceptance probability P. Otherwise, p' is rejected, and the original arrangement p is retained for the next iteration. The above procedures are repeated until all variations of the compression error are smaller than a threshold δ in K consecutive iterations.

In our experiments, we compute the acceptance probability P of SA using the Boltzmann distribution, $\exp(-\Delta E(p, \tilde{p})/T)$, where T denotes temperature, defined as $k^\alpha T_0$. The two parameters k and α control the decreasing speed of T. We empirically set $k = 0.991$, $\alpha = n/10$, where n is the number of iterations, and initialize T_0 with the average error value $E(p^0)$ of the initial arrangement p^0 in Eq. 2. The ending criterion δ is set to $\frac{1}{2000}E(p^0)$, and K is set to 5000 for stability. Fig. 5 shows that the RMSE value of Fig. 2 decreases gradually during the SA optimization iterations and finally converges.

5.3 Initial Arrangement

An initial guess of the solution is essential to begin the optimization. In our case, the error mainly comes from quantization. Note that the maximum quantization error for each block depends on the largest and smallest values in the block. If all values in a block are bounded in a smaller range, the bound of the quantization errors will also be smaller.

Based on this analysis, we obtain an initial arrangement by greedily grouping the closest vertex data iteratively based on their L^2 distance. The size of each group is equal to the block size B of the selected texture compression technique. First, we select an arbitrary vertex as a reference and assign $B-1$ vertices with shortest distance to this reference for grouping. Then, we select another unassigned vertex as a new reference. In general, this reference vertex can be selected randomly. An alternative method is to select a vertex as the nearest unassigned vertex to the last block. Thanks to the SA optimization process, according to our experiment, the selection does not greatly affect the final result. With this new reference, we can search for another $B-1$ nearest neighbors among the unassigned vertices. This step is repeated until all vertices are assigned to a group.

Note that this greedy search does not guarantee that the optimal arrangement will be obtained. However, it is a good initial guess for the optimization process discussed in earlier sections.

5.4 Rendering with Compressed Texture

To render the compressed data in real-time in the graphics rendering pipeline, we need to first upload the compressed texture to the GPU using texture functions (e.g., *glTexImage2D()*). Then, we composite the 2D texture coordinate (u, v) of the facelist to 1D data ($v \times width + u$) and store it in the generic vertex attribute arrays using *glBufferData()*. This composition step allows us to pack the texture coordinate into one 4-byte integer variable of the vertex array in OpenGL. Then, *glVertexAttribPointer()* can be used to bind the arrays into the variables of the vertex shader. By invoking the rendering function (e.g., *glDrawArrays()*), we can specify the geometric primitives in the facelist. For instance, we can specify each triplet of the elements in the facelist as a triangle. Subsequently, a vertex shader is used to access the uploaded texture with the coordinate stored in the facelist. The decompression of the texture is performed directly within the texture lookup functions of the shader (*texture2D()*). Listing 1 shows an example of our vertex shader that handles the compression of the vertex position. It is similar to the shader (Listing 2) of traditional rendering. Finally, OpenGL automatically reconstructs the primitives based on the value of the *gl_Position*.

```
in uint index;
uniform uint width;
uniform sampler2D tex;
void main() {
    vec2 coord = { index % width, index / width };
    gl_Position = MVPMatrix * texture2D(tex,coord);
}
```

Listing 1: The vertex shader of our method.

```
in vec3 vertex;
void main() {
   gl_Position = MVPMatrix * vertex;
}
```

Listing 2: The vertex shader of traditional rendering.

5.5 GPU Implementation

Our SA optimization process is fully implemented on the GPU. It is particularly simple and efficient to speed up by parallelism in our case because our objective function is based on an aggregation of compression errors from independent blocks. More specifically, for each iteration in our optimization, we first randomly group all blocks into pairs and then randomly select a pair of pixels in each group for swapping (Fig. 6). The error metric is evaluated to determine if the swapping is accepted or rejected in each group individually. Because the swapping occurs only to a specific block pair, the operation is independent and well suited for parallelization. To maximize the scale of parallel processing, we set the block size B in our optimization exactly as in the employed texture compression technique because B is also the minimum block size at which swapping affects the error metric. Hence, the maximum number of parallel groups $\frac{\#block}{2}$ can be achieved. In this manner, we improve the efficiency of the SA process significantly.

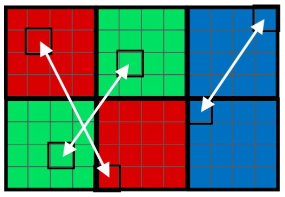


Fig. 6: Parallel implementation of our method. We first randomly group blocks into pairs (marked in the same colors) and randomly swap a pair of elements for each group in parallel.

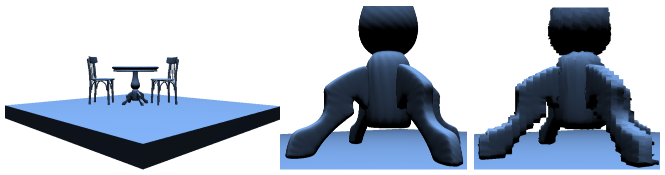

(a) Original model (b) Zoom-in of (a) (c) Compressed

Fig. 7: “Teapot in a stadium” problem. A small model with very fine details in a big scene.

6 Vertex Clustering

Although our permutation method drastically reduces the compression error, the quantization involved still produces errors and affects the visual quality in the “Teapot in a stadium” problem. This occurs when a small model with fine details is placed inside a relatively large scene. As an example, a table set (Fig. 7) with very minute details is placed on top of a large coarse platform. When quantization is applied on this model, most of the details on the table are lost. To address this problem, we further propose a clustering approach.

Depending on the nature of the scene and application, one can choose any clustering method to group the vertices. In this paper, we use the simplest clustering method, k-means clustering, to demonstrate that even the simplest method can provide significant improvement of visual quality in the final results.

The distance used for clustering is the L^2 distance of the input vertex data (Fig. 8). Data in different clusters can be packed into the same texture. Note that swapping is allowed only between blocks in the same cluster. This avoids mixing vertices with different scales in the same block and reduces the possibility of serious quantization error. Fig. 12(d) shows the results of compressing meshes in the “Teapot in a stadium” problem using our method of clustering.

Fig. 9 plots the compression error of Fig. 12 during the optimization process with different numbers of clusters, such as 1, 2, and 8. The error is reduced when the number of clusters is increased. This improvement affects the visual quality of the

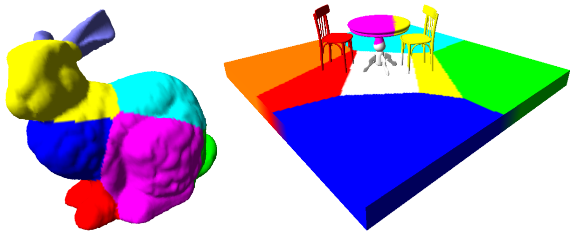


Fig. 8: Vertex clusters on 3D models.

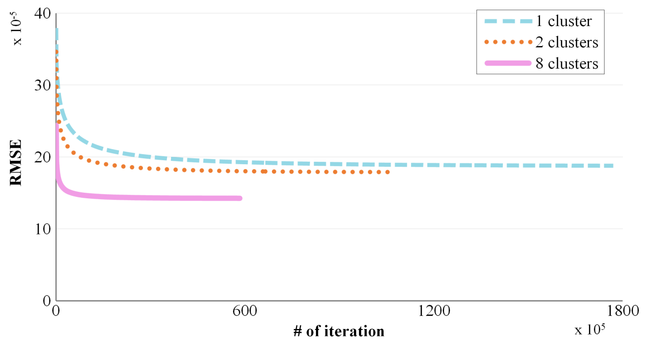


Fig. 9: The compression error plot of Fig. 12 with different numbers of clusters during SA optimization.

compressed models with limited cost and accelerates the optimization step because the number of possible swaps is limited by the clusters.

Note that this vertex clustering is performed on the vertex data only. The connectivity of the 3D model is not affected. Hence, we can avoid strong artifacts such as cracks between clusters.

Because the number of clusters is usually small (not more than 16), we can represent the cluster label using only 4 bits. To extend Listing 1 for supporting clusters, we can first composite the 2D texture coordinate (u, v) and the cluster label (c) to 1D data $((v \times width + u) \times clusterNum + c)$. This allows us to pack all necessary labels into one 4-byte integer variable. Then, we upload the scales and the offsets of the clusters to the vertex shader. Listing 3 shows the extended version of the vertex shader for handling the compressed position.

```
in uint index;
uniform uint width;
uniform uint cluster_Num;
uniform vec3 scale[clusterNum];
uniform vec3 offset[clusterNum];
uniform sampler2D tex;
void main() {
   uint c_Id = index % clusterNum;
   uint index2 = index / clusterNum;
   vec2 coord = { index2 % width, index2 / width };
   gl_Position = MVPMatrix * (texture2D(tex,coord) * ↩
        scale[c_Id] + offset[c_Id]);
}
```

Listing 3: The vertex shader of our method with clustering.

7 Results and Discussion

To thoroughly verify the capability and effectiveness, a series of experiments is performed to test our proposed method in various perspectives. These experiments include evaluations of the rendering quality or time performance when applying our method to different vertex attributes, different coordinate systems, large-scale models, and different hardware texture compressions.

For comparison with existing methods without bias, we seek compression techniques that can preload compressed data to the GPU memory and perform real-time decompression solely on the GPU within the graphic rendering pipeline. To the best of our knowledge, only the methods proposed by Purnomo et al. [1] and Lee et al. [25] fulfill these requirements. Thus, we conducted experiments to compare their method with ours. All experiments were conducted on a PC equipped with an Intel Core i7-6700 3.4GHz CPU, 16 GB RAM, and an nVidia GeForce GTX780 GPU.

7.1 Comparison with Other Methods

In the first experiment, we apply our method to several commonly used vertex attributes, including positions, normals, and texture coordinates. In each case, we compress the mesh with or without permutation and compare the rendering results. We also include the corresponding results of [1]. To permit fair comparisons, we consistently use 12 bpv for 3D data and 8 bpv for 2D data for compression for all methods shown. The bit allocated for [1] is (5:5:2) in Fig. 2, with 4 bits for each dimension in Figs. 10-13.

Fig. 2 and 12 show the rendering results when the position data are compressed. In addition to the position data, we compress the normals (Fig. 10) and texture coordinates (Fig. 11) for various models. For Fig. 13, we compress both position data and normal data with a single facelist. From the experimental results, it is obvious that our proposed approach outperforms the previous method [1] for both 12 bpv and 8 bpv. One possible explanation for the poor results of the previous method is that the precision of geometry is insufficient to handle the “Teapot in a stadium” problem.

Next, we compare our method with [25]. As our method does not have the flexibility to choose the precision of geometry or bit per vertex because of the limitation of hardware compression, we compare our method with [25] in two different cases. We manually select the number of their partitions to obtain results with (i) comparable distortion in terms of RMSE and (ii) comparable size in terms of memory. Here, we use 12 bpv for 3D data and 8 bpv for 2D data for quantization. Fig. 15 shows the comparison in these two cases. Although [25] can obtain results with similar visual quality as ours using a large number of partitions, each partition consumes 16 bytes of extra memory and thus requires 10% to 300% more memory space than our method. When there are fewer partitions, their visual quality is lower than ours. Most importantly, hardware decompression is faster than decompression in a shader. Thus, our proposed approach still outperforms their method.

The majority of existing geometry compression methods solely target a single manifold mesh. Multiple meshes must be handled separately. By contrast, our method can handle scenes with multiple disconnected meshes simultaneously. To demonstrate this capability, we render different scenes with the position data and normal data compressed with a single facelist. Fig. 12 and Fig. 14 show the rendering results for scenes using our method.

7.2 Performance Statistics

Table 1 presents all statistical data for the performance of our method collected from our experiments. Our method achieves a compression ratio of 1:7 to 1:8 (because of the overhead). Note that the data size in the table is the size of the vertex data preloaded in the GPU memory. Although the facelists are also cached inside the GPU memory by OpenGL, the size of the facelist is not included because the main focus of our study

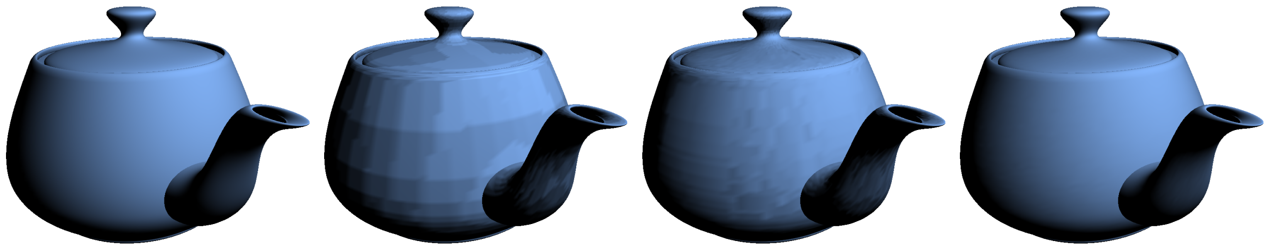

(a) Original (b) Purnomo et al. [1] (c) Without permutation (d) Our result

Fig. 10: The rendering results for a 3D model with normal data compressed. All methods here use 12bpv.

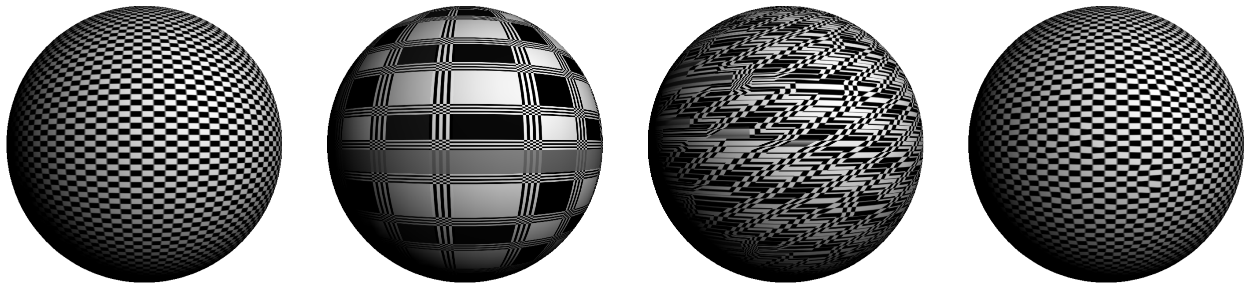

(a) Original (b) Purnomo et al. [1] (c) Without permutation (d) Our result

Fig. 11: The rendering results for a 3D model with texture coordinate compressed. All methods here use 8bpv.

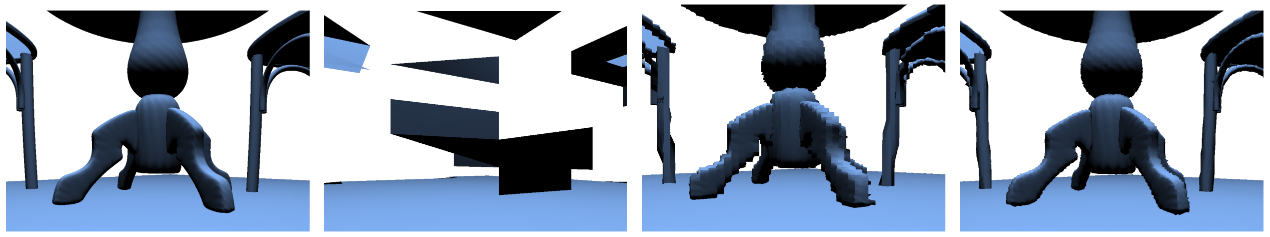

(a) Original (b) Purnomo et al. [1] (c) Without permutation (d) Our result with 8 clusters

Fig. 12: The rendering results for a 3D model with position data compressed. All methods here use 12bpv.

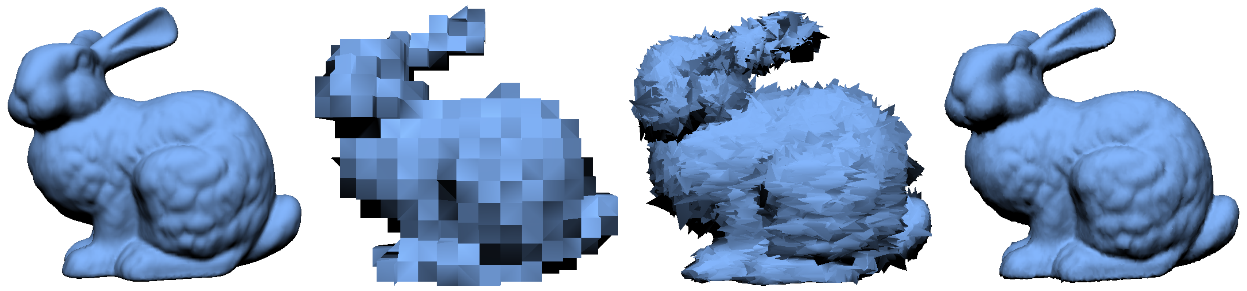
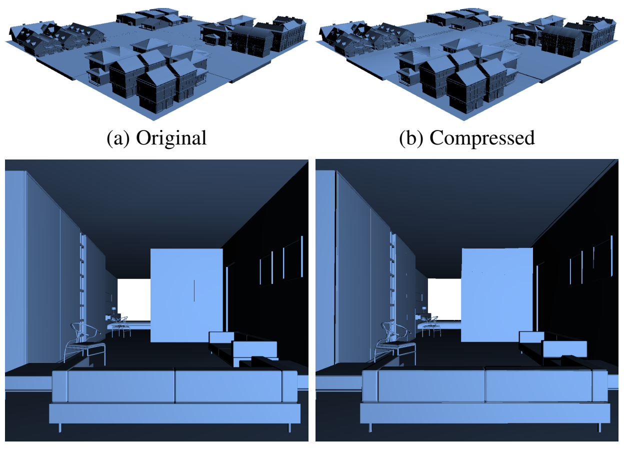

(a) Original (b) Purnomo et al. [1] (c) Without permutation (d) Our result

Fig. 13: The rendering results for a 3D model for which position and normal data are compressed. All methods here use 24 bpv.


(a) Original (b) Compressed

(c) Original (d) Compressed

Fig. 14: The rendering results of scenes with compressed data.

is the compression of vertex data. For compression quality, in addition to a visual comparison of the rendering results, we employ the RMSE metric for the attribute data and peak signal-to-noise ratio (PSNR) metric for the rendered data. To measure PSNR, we sample the rendering results from different viewing angles and average them; all background pixels are ignored. Our method clearly achieves much lower RMSE and much higher PSNR values than compression without permutation (marked as naïvely in Table 1).

In terms of efficiency, the rendering speed is obviously increased by the traditional preloading data method (storing data in VBO). Our permutation takes several minutes for a model, although it saves the memory resources of the GPU without significantly affecting its rendering speed.

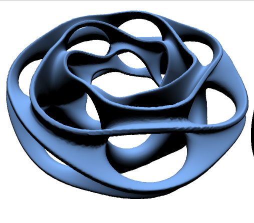
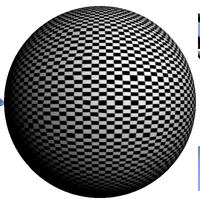
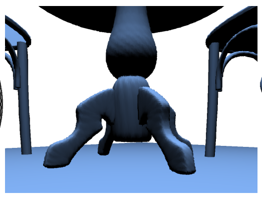
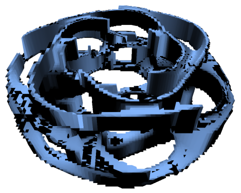
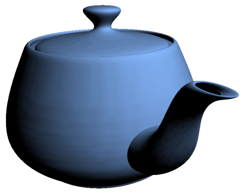
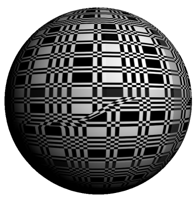
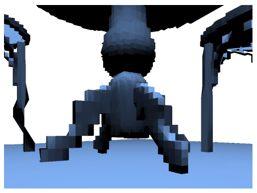
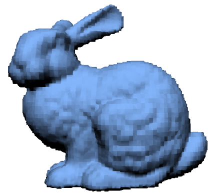
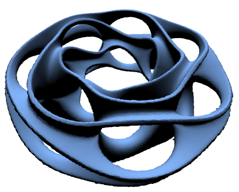
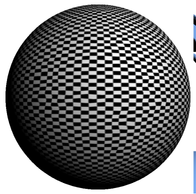
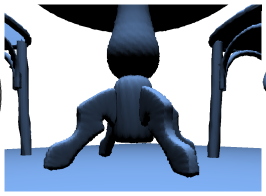
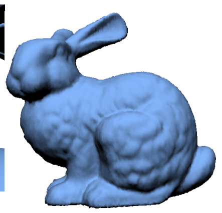
	Fig. 2	Fig. 10	Fig. 11	Fig. 12	Fig. 13
Similar error (RMSE) Lee et al. [25]	 9000 partitions 561.15KB	480 partitions 59.39KB	 300 partitions 6.14KB	 6000 partitions 322.56KB	7000 partitions 324.06KB
Similar size Lee et al. [25]	 16 partitions 419.84KB	 32 partitions 52.22KB	 4 partitions 3.07KB	 32 partitions 228.35KB	 32 partitions 105.47KB
Our results	 419.84KB	52.22KB	 3.07KB	 228.35KB	 105.47KB

Fig. 15: Comparison of the rendering results and memory consumed in [25] and our method. The partition numbers of [25] were manually selected to obtain comparable distortion (first row) or memory size (second row) to permit a fair comparison.

TABLE 1: Statistics of our method for various models

Figure no.	Fig. 2	Fig. 10	Fig. 11	Fig. 12		Fig. 13
Attribute type	Position	Normal	Texture Coord.	Position		Position + Normal
# of element	286,678	35,305	2,650	156,013		35,947
# of dimension	3	3	2	3		6
Original vertex data size	3359.74KB	413.70KB	20.48KB	1827.84KB		842.75KB
# of cluster	1	1	1	1	8	1
RMSE (Initial)	2.39×10^{-6}	1.21×10^{-5}	4.76×10^{-5}	3.79×10^{-5}	2.44×10^{-5}	4.01×10^{-7}
RMSE (Final)	1.84×10^{-6}	8.34×10^{-6}	1.20×10^{-5}	2.00×10^{-5}	1.46×10^{-5}	2.88×10^{-7}
Time for permutation	8.4mins	2.2mins	0.2mins	27.8mins	8.9mins	1.0mins
FPS (Traditional rendering)	17.85	147.36	1572.57	32.45		175.19
FPS (Preloading)	970.87	4178.85	6850.49	2558.09		4100.28
FPS (Ours)	966.18	4096.68	6732.87	2305.92	2292.70	4022.53
Compressed size ([1])	419.84KB	52.22KB	3.07KB	228.35KB		105.47KB
Compressed size ([25]-Similar error)	561.15KB	59.39KB	6.14KB	243.71KB	322.56KB	324.06KB
Compressed size ([25]-Similar size)	419.84KB	52.22KB	3.07KB	228.35KB	228.35KB	105.47KB
Compressed size (Ours)	419.84KB	52.22KB	3.07KB	228.35KB	228.35KB	105.47KB
Bit per vertex (Ours)	12	12	8	12	12	24
PSNR ([1])	10.05	40.81	13.47	8.02		20.02
PSNR ([25]-Similar error)	24.62	59.93	23.06	16.34	16.37	40.97
PSNR ([25]-Similar size)	12.28	50.56	13.33	9.51	14.53	25.87
PSNR (Naïvely)	11.87	43.87	13.43	15.04		17.76
PSNR (Ours)	23.12	57.81	23.24	17.73	20.93	31.62

7.3 Data with Different Coordinate Systems

Although our error metric (Eq. 2) is defined in a Cartesian coordinate system, our method can be directly applied to the compression of data in other coordinate systems, such as polar coordinates for position data and spherical coordinates for normal data. Fig. 16 shows the rendering results for a compressed mesh under different coordinate systems. To compress the toroids positional data, the quality of the result in Cartesian coordinates (Fig. 2) is comparable to that of its counterpart in polar coordinates (Fig. 16(a)). However, when applied to normal compression, the use of spherical coordinates added slight noise to the result because 3D data are stored in 12 bpv, whereas 2D data are stored in 8 bpv only. Thus, the compression quality is lower when a 2D coordinate system is used, although the difference is not visually apparent.

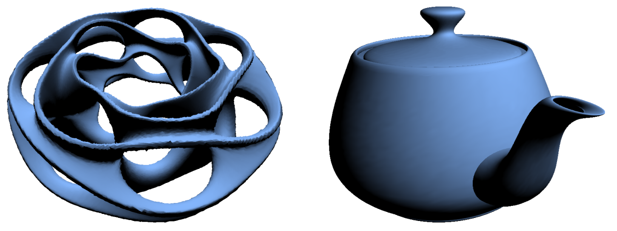

(a) Polar coordinate (Position) (b) Spherical coordinate (Normal)

Fig. 16: Rendering results for compressed data with different coordinate systems.

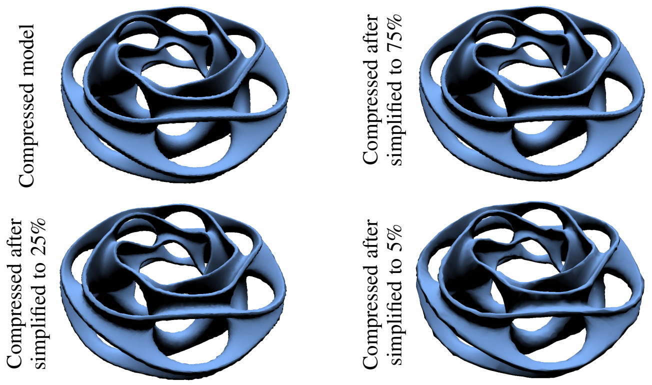

Compressed model
Compressed after simplified to 75%
Compressed after simplified to 25%
Compressed after simplified to 5%

Fig. 17: Our method can accept simplified models with various degrees of simplification as input and obtain acceptable rendering results after compression.

7.4 Compression of Connectivity Data

Although this paper mainly focuses on the compression of vertex data, the connectivity data should not be ignored. Here, we provide two optional methods to reduce the size of connectivity data.

The first approach is to use mesh simplification, which demonstrates that simplification can be used to reduce the size of topology data effectively [1]. We employ the idea given in [1] and simplify the model before our compression. Fig. 17 shows the rendering results when our compression is applied to models with different degrees of simplification. Our method can effectively take the simplified model as input and obtain reasonable quality.

The second approach is to use triangle stripification, a well-known compression approach in MPEG-4 standard. Stripification minimizes the connectivity data by representing N triangles ($3N$ vertices) with $3 + (N - 1)$ vertices losslessly. As a result, the size of the facelist is reduced by approximately 3 times. It is an effective method to reduce the size of connectivity data in the GPU memory. Most importantly, rendering of triangle strips is directly supported by OpenGL, although it may not work well for nonmanifold meshes or a polygon soup.

7.5 Compression of a Large-Scale Model

Our proposed mesh compression approach is especially suitable for complex and large-scale 3D models because they may easily reach the limits of GPU memory and processing power. Thus, we conducted two experiments with large-scale mesh models (Fig. 18) to evaluate the performance of our method. In the first experiment, we compressed a large 3D model (merlion) containing more than 13.5 million vertices. The corresponding data size in memory is nearly 414.9 MB. Without any compression, this model can be rendered in 32.3 fps using the preloading method. With the use of our compression method, an average frame rate of 30.4

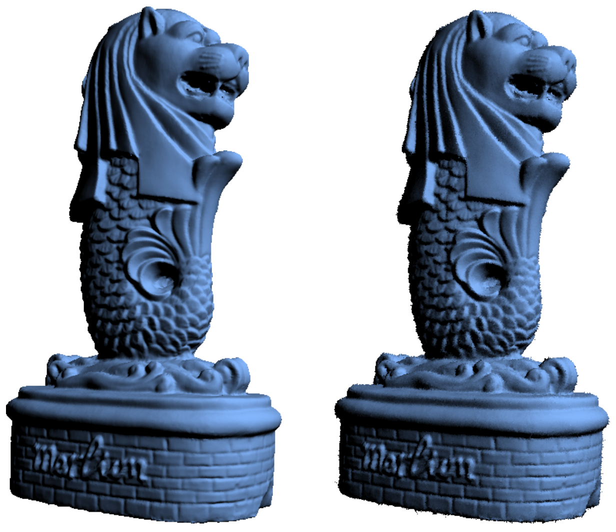

(a) Original (b) Compressed

Fig. 18: The rendering results of our method for a large 3D model

fps is maintained, while the memory consumption is reduced to 51.9 MB. Thus, our method reduces GPU memory consumption without sacrificing rendering speed even on large models.

The second experiment investigates the performance trend of different rendering methods when the number of polygons increases. We rendered a standard sphere in different resolutions using various approaches, including traditional rendering, the preloading method, and our method. Fig. 19 shows the rendering performance for various combinations of mesh size and methods in terms of fps. Traditional rendering (blue dash-dot line) is slowest, as it requires transmitting the vertex data from the CPU to the GPU 30 times per second. However, it does not place any demands on the GPU memory size.

Preloading data to the GPU (green dotted line) improves the rendering speed compared to transmitting data every time. The drawback is the limited size of GPU memory, which cannot accommodate vertex data, especially for a large, complicated scene. In this experiment, there is 1.5 GB of available memory on the GPU. The preloading method could not further process data with a size larger than 1.5 GB, as shown in the figure. In other words, we can only preload a model with less than 49 million vertices. By contrast, our method (red thick line) can effectively reduce the preloaded data size and render the data without significantly affecting the rendering speed. Even a model with 100 million vertices can be preloaded to the GPU.

7.6 Compression with Various HW Techniques

All of the results of our above-mentioned method were compressed using 3Dc. However, our method is not limited to the use of 3Dc but can support different block-based hardware compression techniques such as S3TC (4bpv) and BPTC (8bpv). However, because of the difference in geometric precision, the resultant compression qualities vary as shown in Fig. 20. Our method achieves better results when a method with higher geometric precision is applied.

8 Conclusions

In this article, we propose a novel method to enable storing of compressed vertex data in GPU memory and full decompression

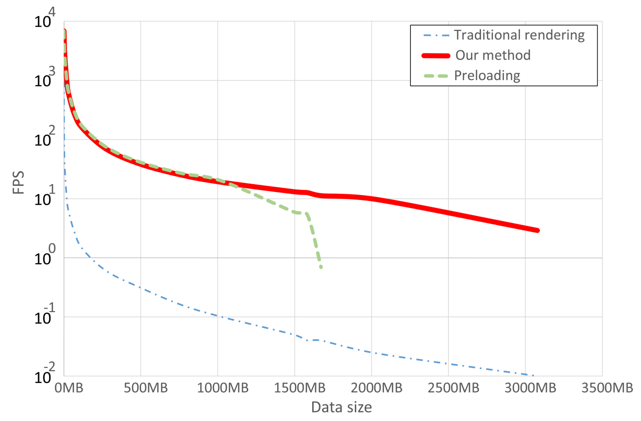


Fig. 19: Performance trend of rendering speed for models with different complexities using traditional rendering, the preloading method, and our method. We plotted log-transformed performance (in units of fps) against the data size (in units of MB).

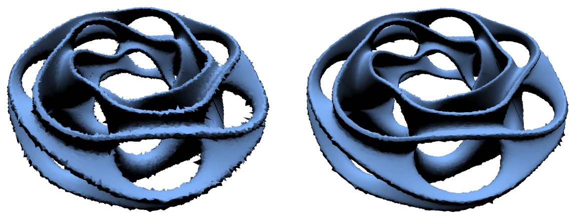


Fig. 20: Our method supports different block-based hardware texture compression methods.

on the GPU during real-time rendering. This allows us to regain memory space on the GPU to store more data without sufficiently affecting the rendering speed. The key idea is to exploit existing texture compression techniques embedded inside ordinary graphics hardware for vertex data.

However, naïvely compressing the vertex data using texture compression techniques may not obtain an acceptable result because the coherence of the vertex data in textures is not guaranteed. Therefore, our key contribution is the proposal of a number of methods to increase data coherence. To minimize the compression error, our proposed method permutes the vertex data by optimization. Moreover, we further reduce the compression error by performing vertex clustering. Our experimental results demonstrate that our proposed method is efficient and effective.

Our method is designed to be compatible with different block-based hardware texture compression techniques, such as S3TC, 3Dc, and BPTC. We believe that our method is sufficiently general for different types of vertex data attributes such as color, material, and BRDF.

There are three limitations of our method. First, our method relies on the compression circuit within the graphics hardware. Thus, flexibility in choosing the bitrate, compression ratio, and performance of our method is limited by the hardware configurations. Latest texture compression techniques (e.g., Adaptive Scalable Texture Compression (ASTC) [39]) cannot be applied without hardware support on ordinary GPUs. Second, the texture compression method is often lossy and may generate high-frequency noise that is more noticeable than low-frequency error. In the future, we intend to address this problem with real-time geometry filtering on a GPU. Third, our method may not work well for multiple vertex attributes that are not highly correlated. Although it is possible to use different permutations for multiple attributes using multiple generic vertex attribute arrays, the trade-off is the increase in the size of the facelist.

As this paper mainly focuses on compression of vertex data, we intend to further investigate methods for hardware connectivity data compression in the future. Furthermore, we intend to extend our method to support animated 3D models. The challenge here is how to exploit temporal coherence from the vertex data among frames so that a higher compression ratio can be achieved with the use of hardware 3D texture compression.

Last but not least, we intend to further investigate different mesh distortion metrics [40] for better error measurement in our optimization.

Acknowledgments

We thank the reviewers for the valuable comments. This work is supported by Shenzhen Science and Technology Program (No.JCYJ20160429190300857), Research Grants Council of the Hong Kong Special Administrative Region, under RGC General Research Fund (Project No. 14200915), Research Grants Council of the Hong Kong Special Administrative Region, China (Project Reference No. UGC/FDS11/E03/14), and Special Fund of Science and Technology Research and Development on Application From Guangdong Province (Grant No. 2016B010124011).

References

[1] B. Purnomo, J. Bilodeau, J. D. Cohen, and S. Kumar, “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: ACM, 2005, pp. 53–61.

[2] A. Khodakovsky, P. Schröder, and W. Sweldens, “Progressive geometry compression,” in *Siggraph 2000, Computer Graphics Proceedings*, K. Akeley, Ed. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000, pp. 271–278.

[3] R. Pajarola and J. Rossignac, “Compressed progressive meshes,” *IEEE Trans. Vis. Comput. Graph.*, vol. 6, no. 1, pp. 79–93, 2000.

[4] P. Alliez and M. Desbrun, “Progressive compression for lossless transmission of triangle meshes,” in *SIGGRAPH ’2001 Conference Proceedings*, 2001, pp. 198–205.

[5] J. Kessenich, D. Baldwin, and R. Rost, *The OpenGL Shading Language*, June 2010.

[6] I. NVIDIA, *NVIDIA CUDA C Programming Guide*, 2012.

[7] M. Deering, “Geometry compression,” *Computer Graphics*, vol. 29, no. Annual Conference Series, pp. 13–20, 1995.

[8] M. M. Chow, “Optimized geometry compression for real-time rendering,” in *VIS ’97: Proceedings of the 8th conference on Visualization ’97*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997, pp. 347–ff.

[9] C. Touma and C. Gotsman, “Triangle mesh compression,” in *Graphics Interface*, 1998, pp. 26–34.

[10] M. Isenburg, “Compressing polygon mesh connectivity with degree duality prediction,” in *Proc. Graphics Interface*, May 2002, pp. 161–170.

[11] L. Vasa and G. Brunnett, “Exploiting connectivity to improve the tangential part of geometry prediction,” *IEEE transactions on visualization and computer graphics*, vol. 19, no. 9, pp. 1467–1475, 2013.

[12] X. Hao and A. Varshney, “Variable-precision rendering,” in *I3D ’01: Proceedings of the 2001 symposium on Interactive 3D graphics*. New York, NY, USA: ACM, 2001, pp. 149–158.

[13] X. Gu, S. J. Gortler, and H. Hoppe, “Geometry images,” in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH ’02. New York, NY, USA: ACM, 2002, pp. 355–361.

[14] E. Praun and H. Hoppe, “Spherical parametrization and remeshing,” in *ACM SIGGRAPH 2003 Papers*, ser. SIGGRAPH ’03. New York, NY, USA: ACM, 2003, pp. 340–349.

[15] M. B. Rodríguez, E. Gobbetti, F. Marton, and A. Tinti, "Compression-domain seamless multiresolution visualization of gigantic triangle meshes on mobile devices," in *Proceedings of the 18th International Conference on 3D Web Technology*. ACM, 2013, pp. 99–107.

[16] Z. Karni and C. Gotsman, "Spectral compression of mesh geometry," in *Siggraph 2000, Computer Graphics Proceedings*. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000, pp. 279–286.

[17] R. Schnabel, S. Mser, and R. Klein, "A parallelly decodeable compression scheme for efficient point-cloud rendering," in *Proceedings symposium on point-based graphics*, 2007, pp. 119–128.

[18] Q. Meyer, J. Süßmuth, G. Sußner, M. Stamminger, and G. Greiner, "On floating-point normal vectors," in *Computer Graphics Forum*, vol. 29, no. 4. Wiley Online Library, 2010, pp. 1405–1409.

[19] E. Gobbetti, F. Marton, M. B. Rodriguez, F. Ganovelli, and M. Di Benedetto, "Adaptive quad patches: an adaptive regular structure for web distribution and adaptive rendering of 3d models," in *Proceedings of the 17th international conference on 3D web technology*. ACM, 2012, pp. 9–16.

[20] B. Jovanova, M. Preda, and F. Preteux, "Mpeg-4 part 25: A generic model for 3d graphics compression," in *2008 3DTV Conference: The True Vision-Capture, Transmission and Display of 3D Video*. IEEE, 2008, pp. 101–104.

[21] K. Mamou, T. Zaharia, and F. Prêteux, "Tfan: A low complexity 3d mesh compression algorithm," *Computer Animation and Virtual Worlds*, vol. 20, no. 2-3, pp. 343–354, 2009.

[22] M. Luffel, T. Gurung, P. Lindstrom, and J. Rossignac, "Grouper: A compact, streamable triangle mesh data structure," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 1, pp. 84–98, 2014.

[23] D. Calver, "Vertex decompression in a shader," in *ShaderX: Vertex and Pixel Shader Tips and Tricks*, 2002, pp. 172–187.

[24] ——, "Using vertex shaders for geometry compression," in *ShaderX2: Shader Programming Tips & Tricks with DX9*, 2004, pp. 3–12.

[25] J.-S. Lee, S.-Y. Choe, and S.-Y. Lee, "Compression of 3d mesh geometry and vertex attributes for mobile graphics," *Journal of Computing Science and Engineering*, vol. 4, no. 3, pp. 207–224, 2010.

[26] A. Maglo, G. Lavoué, F. Dupont, and C. Hudelot, "3d mesh compression: Survey, comparisons, and emerging trends," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, p. 44, 2015.

[27] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-oblivious mesh layouts," in *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3. ACM, 2005, pp. 886–893.

[28] M. Tchiboukdjian, V. Danjean, and B. Raffin, "Binary mesh partitioning for cache-efficient visualization," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 16, no. 5, pp. 815–828, 2010.

[29] I. ATI, *Radeon X800: 3Dc White Paper. Tech. rep., 2005.*, 2005.

[30] I. S3, *S3TC: White paper*, 1999.

[31] *ARB Texture Compression Bptc*, 2010.

[32] J. E. Lengyel, "Compression of time-dependent geometry," in *Proceedings of the 1999 symposium on Interactive 3D graphics*, ser. I3D '99. New York, NY, USA: ACM, 1999, pp. 89–95.

[33] J. Chhugani and S. Kumar, "Geometry engine optimization: cache friendly compressed representation of geometry," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ser. I3D '07. New York, NY, USA: ACM, 2007, pp. 9–16.

[34] P. Lindstrom and G. Turk, "Image-driven simplification," *ACM Transactions on Graphics*, vol. 19, no. 3, pp. 204–241, 2000.

[35] M. Garland and P. S. Heckbert, "Simplifying surfaces with color and texture using quadric error metrics," in *IEEE Visualization '98*, D. Ebert, H. Hagen, and H. Rushmeier, Eds., 1998, pp. 263–270.

[36] H. H. Hoppe, "New quadric metric for simplifying meshes with appearance attributes," in *IEEE Visualization '99*, D. Ebert, M. Gross, and B. Hamann, Eds., San Francisco, 1999, pp. 59–66.

[37] R. E. Smith and D. Holtkamp, "A representation for permutation optimization with a combinatorial genetic algorithm," 2007.

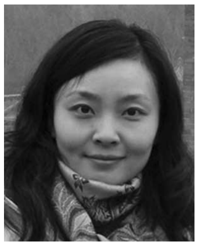
[38] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.

[39] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson, "Adaptive scalable texture compression," in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Eurographics Association, 2012, pp. 105–114.

[40] M. Corsini, M.-C. Larabi, G. Lavoué, O. Petřík, L. Váša, and K. Wang, "Perceptual metrics for static and dynamic triangle meshes," in *Computer Graphics Forum*, vol. 32, no. 1. Wiley Online Library, 2013, pp. 101–125.

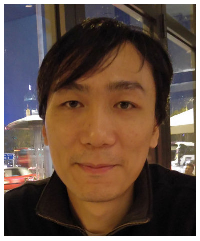
Kin Chung Kwan Kin Chung Kwan (KC) is now a research fellow at the School of Computing and Information Sciences in Caritas Institute of Higher Education, Hong Kong. He received his B.Sc., and Ph.D. degree in the Department of Computer Science and Engineering from The Chinese University of Hong Kong in 2009 and 2015 respectively. His research interests include computer graphics, real-time rendering, non-photorealistic rendering, GPGPU, and shape analysis.

Xuemiao Xu Xuemiao Xu received her B.S. and M.S. degrees in Computer Science and Engineering from South China University of Technology in 2002 and 2005 respectively, and Ph.D. degree in Computer Science and Engineering from The Chinese University of Hong Kong in 2009. She is currently a professor in the School of Computer Science and Engineering, South China University of Technology. Her research interests include object detection&tracking&recognition, and image&video understanding and synthesis.



Liang Wan Liang Wan received the B.Eng and M.Eng degrees in computer science and engineering from Northwestern Polytechnical University, P.R. China, in 2000 and 2003, respectively. She obtained a Ph.D. degree in computer science and engineering from The Chinese University of Hong Kong in 2007. She is currently an Associate Professor in the School of Computer Software, Tianjin University, P. R. China. Her research interest is mainly on intelligent image synthesis, including image-based rendering, image navigation, pre-computed lighting, and panoramic image processing.

Tien-Tsin Wong Tien-Tsin Wong received his B.Sc., M.Phil. and Ph.D. degrees in Computer Science from the Chinese University of Hong Kong in 1992, 1994, and 1998 respectively. He is currently a professor in the Department of Computer Science and Engineering, the Chinese University of Hong Kong. His main research interests include computer graphics, computer vision, computational perception, computational manga, GPU techniques and image-based rendering. He received the IEEE Transactions on Multimedia Prize Paper Award 2005 and the Young Researcher Award 2004.



Wai-Man Pang Wai-Man Pang (Raymond) is now an associate professor at the School of Computing and Information Sciences in Caritas Institute of Higher Education, Hong Kong. He was with the Computer Graphics Lab, University of Aizu, Japan from 2009 to 2011 as an assistant professor. He finished his postdoctoral fellowship and Ph.D study at the Department of Computer Science and Engineering at CUHK. His current research interests are two folds, one is on graphics related techniques and the other is healthcare related applications. Topics include texture analysis, vision based recognition, image feature extraction, computational manga, hardware accelerated algorithms and health care technologies on mobile devices.