Mesh Geometry Compression for Mobile Graphics

Jongseok Lee
POSTECH
thirdeye@postech.ac.kr

Sungyul Choe
Samsung Electronics
sungyul.choe@samsung.com

Seungyong Lee
POSTECH
leesy@postech.ac.kr

***Abstract*—This paper presents a compression scheme for mesh geometry, which is suitable for mobile graphics. The main focus is to enable real-time decoding of compressed vertex positions while providing reasonable compression ratios. Our scheme is based on local quantization of vertex positions with mesh partitioning. To prevent visual seams along the partitioning boundaries, we constrain the locally quantized cells of all mesh partitions to have the same size and aligned local axes. We propose a mesh partitioning algorithm to minimize the size of locally quantized cells, which relates to the distortion of a restored mesh. Vertex coordinates are stored in main memory and transmitted to graphics hardware for rendering in the quantized form, saving memory space and system bus bandwidth. Decoding operation is combined with model geometry transformation, and the only overhead to restore vertex positions is one matrix multiplication for each mesh partition.**

***Index Terms*—Mesh compression, Geometry encoding, Mobile graphics, Local quantization, Mesh partitioning.**

I. Introduction

Recent mobile devices, such as mobil phones, PDAs, and hand-held game players, are equipped with quite decent features of computing power, storage, graphics, and network. Especially, for mobile graphics, API standards such as OpenGL ES and JSR-184 have been proposed [8], and the graphics performance has been drastically improved with development of mobile GPUs. However, compared to desktop PC graphics, mobile graphics is still constrained by limited resources of low computing powers, small amounts of memory, and low bandwidths. In addition, while mobile devices are usually powered by batteries, battery technology has not caught up with the energy requirement of 3D graphics processing.

Data compression can provide an excellent solution to help overcome these limitations. Obviously, compressed graphics data enable us to better utilize storage space and network bandwidth. Furthermore, by decoding compressed data on-the-fly on graphics hardware [11], we can reduce memory access and data transmission through the system bus, which saves power consumption. However, to be suitable for mobile graphics, a compression technique should be simple enough not to impose a big overhead on the low computing power.

In this paper, we propose a simple and effective compression technique for mesh geometry, which can be used for mobile graphics in practice. The main design objective is real-time on-the-fly decoding of vertex positions on graphics hardware. Such property enables the compressed form to be used for both storing the vertex positions of a mesh in main memory and transmitting to graphics hardware for rendering. As a result, main memory space and system bus bandwidth required for mesh geometry can be reduced in a rendering system.

To fulfill the objective, we use local quantization of vertex positions with mesh partitioning. However, straightforward local quantization that processes each mesh partition separately suffers from visual seams along the partitioning boundaries of the restored mesh. To prevent the problem, we quantize all mesh partitions in a coherent way to have the same-sized and axis-aligned quantized cells. Then, the distortion of the restored mesh relates with the size of the quantized cells and we propose a mesh partitioning algorithm to minimize the size.

The setting of our experiments is to quantize a 32-bit floating point vertex coordinate into an 8-bit integer, which is the smallest data size supported in a mobile graphics library. Local quantization incurs encoding overhead for bounding cube information and duplicated vertices. However, experimental results show that the data reduction ratios are still over 70%. The distortions of the restored meshes are comparable to 11-bit global quantization of vertex coordinates, which is close to 12-bit global quantization that is considered enough for representing the geometry of moderate-sized meshes in mesh compression [1], [7].

II. Related Work

Mesh compression has been an active research area in graphics. A comprehensive review of mesh compression techniques, which is not within the scope of this paper, can be found in excellent survey papers [1], [7].

Although previous mesh compression algorithms offer good performance in compression ratio, their encoding and decoding processes are rather complicated and can impose much overhead on the relatively low computing power of mobile hardware. More importantly, most of them sequentially traverse mesh elements for encoding, which prohibits parallel decoding of encoded elements on graphics hardware. Consequently, previous algorithms cannot be directly adopted for mesh compression on mobile devices.

Calver [2] described the basic principles for various quantization methods of vertex geometry which allow real-time decoding with programmable vertex shaders. He mentioned that local quantization can be used for vertex geometry to obtain better compression rate-distortion. However, he presented no solution for the problem of visual seams and no specific technique to partition a given mesh for local quantization.

Purnomo *et al.* [9] refine the idea of Calver and allocate different numbers of bits for the components of vertex attributes, such as positions, normals, and texture coordinates, to minimize the image-space error. At rendering time, compressed vertex data, which have been packed into fixed 96 bits,

are transmitted to a programmable vertex shader and restored on-the-fly. However, in terms of vertex geometry compression, their technique uses global quantization, which provides worse compression rate-distortion than local quantization.

In contrast to [2], in this paper, we provide effective solutions to handle visual seams and mesh partitioning for local quantization. Compared to [9], in addition to the difference of using local quantization, our technique requires no programmable vertex shader because the decoding operation can be incorporated into the standard geometry transformation, which makes it applicable to a wide range of mobile graphics platforms.

III. Basic Idea

In this paper, we only consider compression of mesh geometry. We assume that mesh connectivity is represented in an adequate form (e.g., triangle strips), which is supported in mobile graphics API standards, such as JSR-184 [8].

To restore vertex geometry in real time on a mobile device, a compression algorithm should satisfy the following properties;

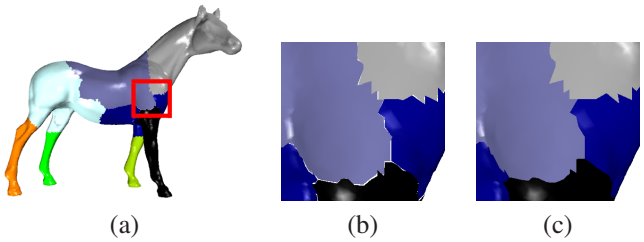
- The decoding process is simple enough for real-time performance.
- The decoding operations of vertex positions have no dependency among each other to allow parallel processing.
- Encoded data are represented in a fixed-size format, which matches with the data channel size between CPU and graphics hardware.

A simple solution to satisfy these requirements is the quantization of vertex positions. Original vertex positions, represented by 32-bit floating point numbers, can be encoded with fewer bits by truncating least significant bits. We can easily restore the original positions from the quantized ones by simple additions and multiplications. In addition, there is no dependency among quantized vertex positions.

In compression of irregular meshes with reasonable sizes, 12-bit global quantization is assumed to introduce no strong visual artifacts and used as a preprocessing step before geometry encoding [1], [7]. However, the use of 12-bit quantized positions can cause unnecessary bitwise operations or waste of bandwidth because most graphics APIs support only fixed-sized data channels (e.g., 8, 16, and 32 bits). On the other hand, if we globally quantize vertex positions with 8 bits, we may have severe distortions of mesh geometry (see Fig. 7(a)).

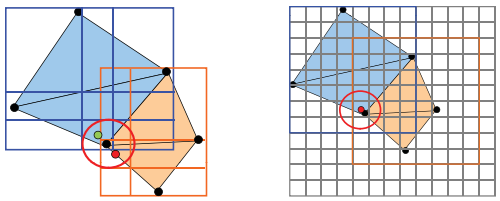
To resolve such a problem, Calver [2] described a local quantization method. A mesh is divided into several sub-meshes and each sub-mesh is independently quantized. Then, the distortion of the restored mesh is reduced because the size of a bounding cube, which is the range of quantization, decreases. As a result, 8-bit local quantization of mesh geometry can provide better accuracy for restored vertex positions than global quantization (see Fig. 7(b)). However, in this case, the restored mesh contains a critical problem, i.e., visual seams along the sub-mesh boundaries, as shown in Fig. III(b). Calver [2] did not provide any solution to handle the problem.

The main cause of the visual seam problem is that shared boundary vertices of sub-meshes should be encoded more than once. Sub-meshes are disjoint subsets of mesh faces and a boundary vertex of a sub-mesh also belongs to one or more other sub-meshes, except at the mesh boundary. When sub-meshes are separately encoded, a boundary vertex v shared among k sub-meshes is encoded k times, once for each sub-mesh, with different local quantization. At decoding time, a restored vertex position is the center of a quantized cell. The restored positions of v in the k sub-meshes do not match each other if the corresponding locally quantized cells of the sub-meshes are not aligned. See Fig. III(a) for illustration.



(a) (b) (c)

Fig. 1. (a) Rendering of a horse model. (b) and (c) show zoom-ins of the red rectangle in (a); (b) Visual seams from straightforward local quantization. (c) Visual seams have been resolved by our method.



(a) analysis (b) our solution

Fig. 2. (a) A black vertex shared by two sub-meshes is restored onto the green and red positions due to the difference of locally quantized cells. (b) The restored positions from two sub-meshes have become the same due to the alignment of locally quantized cells.

The basic idea of our solution for the visual seam problem is to align the locally quantized cells of sub-meshes sharing boundary vertices. As illustrated in Fig. III(b), if the locally quantized cells are aligned, the restored positions of a shared vertex will be the same among the sub-meshes. This constraint of aligned quantized cells should hold for every pair of sub-meshes sharing a boundary vertex. Consequently, the constraint propagates through the adjacency of sub-meshes, and the locally quantized cells of all sub-meshes should have the same size and aligned axes. In this paper, we use the largest bounding cube of sub-meshes to determine the cell size and axes of the aligned local quantization, which is then applied to all sub-meshes. Fig. III(c) shows that the rendering result from our local quantization method does not contain any visual seams.

IV. Encoding Process

The encoding process consists of two parts; mesh partitioning, which divides an input mesh into several sub-meshes, and local quantization of vertex positions, which uses the mesh partitioning result.

A. Mesh partitioning

With quantization of vertex positions, the distortion of a restored mesh from the original is dominated by the size of quantized cells. As described in Sec. III, in our approach, the size of locally quantized cells is determined by the largest bounding cube of mesh partitions. Hence, to minimize the distortion, the largest bounding cube should be made as small as possible.

We adapt the mesh partitioning framework based on *Lloyd's algorithm* [6], which has been successfully used for mesh segmentation [10], [3]. After initial partitioning has been obtained with a given number of partitions, the partitioning result is iteratively updated by repeating two steps; *seed re-computation* and *region growing*. In the seed re-computation step, the seed triangles are repositioned at the centers of partitions for the next update. In the region growing step, faces are added to partitions in the increasing order of distances from the nearest seed faces.

For initial partitioning, we use the *hierarchical face cluster merging* approach [5]. At the beginning, each face is a single partition. We iteratively merge two neighbor partitions into one until the number of partitions is reduced to the desired number. To select the merged pair of partitions at each iteration, we use the cost function defined by

$$F(C_i, C_j) \quad = \quad \max\{x_{ij}, y_{ij}, z_{ij}\}, \tag{1}$$

where x_{ij}, y_{ij}, and z_{ij} are respectively the x, y, and z sizes of the axis aligned bounding cube containing the two partitions, C_i and C_j. By minimizing $F(C_i, C_j)$, we can reduce the bounding cube sizes of resulting mesh partitions.

For seed-recomputation and region growing, we use L_∞ metric to define the distance function. To reposition the seed of a mesh partition, we first compute the center of the bounding cube of the partition, which can be considered as the L_∞ center of the partition. Although we can use the center as the new seed of the partition, we select the nearest face to the center as the new seed face because we use the adjacency of faces in region growing for efficiency. In region growing, the distance of a face f from the seed face of a partition C is defined by

$$F(f, C) \quad = \quad \max\{\delta_x, \delta_y, \delta_z\}, \tag{2}$$

where δ_x is the maximum of the x-distances of three vertices of f from the center of the seed face of C. δ_y and δ_z are defined similarly.

In mesh segmentation techniques [10], [3], it has been demonstrated that the result of Lloyd's algorithm partitions a given mesh into almost equal-sized sub-meshes, where the size is measured in the adopted distance function. Similarly, with L_∞ metric, we can expect that the sub-meshes resulting from Lloyd's algorithm have almost same sizes of bounding cubes because a L_∞ sphere in 3D is a cube. As a result, the largest bounding cube of mesh partitions can be made as small as possible for a given number of partitions. Fig. IV-A shows examples of the final partitioning result.

B. Local quantization

In this paper, we use 8-bit quantization for vertex coordinates. The steps for local quantization can be summarized as follows;

Fig. 3. Mesh partitioning results

1) Calculate the bounding cube for each mesh partition and find the largest x, y, and z bounding cube sizes among all partitions.
2) Calculate (C_x, C_y, C_z), the x, y, and z sizes of the quantized cell, by dividing the largest x, y, and z bounding cube sizes by $(2^8 - 1)$, respectively.
3) Quantize all vertex positions by dividing original vertex coordinates by the quantized cell size (C_x, C_y, C_z) and truncating the fractional values.
4) For each partition, find the minimum quantized coordinates for the x, y, and z axes, and keep the values as the offsets (O_x, O_y, O_z). Then, subtract (O_x, O_y, O_z) from the quantized coordinates of vertices, obtaining the final quantized coordinates (P_x, P_y, P_z) in the range of [0, 255].
5) Save the size of the quantized cell, offsets for partitions, and quantized vertex positions.

Note that for calculating the size of the quantized cell, we use $(2^8 - 1)$, instead of 2^8, to guarantee that the quantized vertex coordinates after offsetting fall in the range of [0, 255].

C. File structure

After local quantization, to restore vertex positions, we need the size of the quantized cell, offsets for partitions, and quantized vertex positions. The size of the quantized cell is stored only once for a mesh and represented by three 32-bit floating point numbers. The offsets for a mesh partition correspond to the quantized coordinates of the origin of the bounding cube of the partition. To represent the offsets, we use 16 to 32-bit integers depending on the mesh size. Since we are using 8-bit local quantization, a quantized vertex position requires 24 bits, i.e., three 8-bit unsigned integers. In addition, we should record the number of partitions and the number of vertices in each partition. These numbers are represented by 16 to 32-bit integers depending on the mesh size. Fig. 4 shows the file structure that contains encoded mesh geometry.

There are two kinds of storage overhead incurred by encoding with local quantization. One is the header overhead, which corresponds to the data in the mesh header and partition headers in Fig. 4. The other is the vertex overhead. As mentioned in Sec. III, the boundary vertices shared among sub-meshes should be duplicated when they are locally quantized with different offsets. Hence, in the encoded mesh geometry, there are more quantized vertex positions than the number of vertices.

V. Decoding Process

When we render a mesh with encoded geometry, we first restore the quantized cell size (C_x, C_y, C_z) from the mesh

Mesh Geometry File

Mesh Header	Partition Header	Vertex Positions	Partition Header	Vertex Positions	...

Mesh Header

Cx	Cy	Cz	# of partitions
4 bytes	4 bytes	4bytes	2-4 bytes

Partition Header

Ox	Oy	Oz	# of vertices
2-4 bytes	2-4 bytes	2-4bytes	2-4 bytes

Vertex Position

Px	Py	Pz
1 byte	1 byte	1 byte

Fig. 4. File structure for encoded mesh geometry: C, O, and P denote the quantized cell size, offset value, and vertex coordinate, respectively.

header. Then, each partition of the mesh is rendered with the following procedure;

1) Restore the offset values (O_x, O_y, O_z) from the partition header.
2) Calculate a 4×4 decoding matrix

$$\mathbf{M_d} = \begin{pmatrix} C_x & 0 & 0 & C_x \cdot O_x \\ 0 & C_y & 0 & C_y \cdot O_y \\ 0 & 0 & C_z & C_z \cdot O_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (3)$$

3) Multiply matrix $\mathbf{M_d}$ by the geometry transformation matrix, e.g., the model-view matrix in OpenGL.
4) Render the polygons in the partition.

In the decoding process, vertex geometry data are transmitted from main memory to graphics hardware in the encoded form, i.e., an 8-bit integer for a vertex coordinate. When we render a polygon, the vertex positions are automatically restored through the geometry transformation matrix which has been modified by the decoding matrix $\mathbf{M_d}$. Note that $\mathbf{M_d}$ is the composition of two matrices, one for adding offsets to locally quantized vertex coordinates in [0, 255] and the other for multiplying the quantized cell size by the quantized coordinates after offsetting to obtain 32-bit floating point coordinates.

Our decoding process is nicely incorporated into the standard rendering pipeline. As a result, at rendering time, the only major overhead is one matrix multiplication per a partition, which can be considered negligible. Although boundary vertices are duplicated among partitions, the number of rendered polygons does not change and the number of vertices processed for rendering remains the same.

VI. Experimental Results

We have implemented and tested our algorithm using a variety versions of Khronos OpenGL graphics APIs [8], such as OpenGL 1.x, OpenGL 2.x, and OpenGL ES 2.x emulator. In OpenGL 1.x and OpenGL 2.x implementations, the decoding matrix $\mathbf{M_d}$ is formed and multiplied by the model-view matrix before rendering each sub-mesh. For OpenGL ES 2.x emulator, $\mathbf{M_d}$ is formed and used when we set up a vertex shader for a sub-mesh. Fig. 5 shows models used in our experiments.

In the experiments, we measure the reduction ratio for vertex geometry data. We also count the number of duplicated vertices along the boundaries of mesh partitions. That is,

$$\textit{Data Reduction Ratio} = \left(1 - \frac{\textit{compressed data}}{\textit{uncompressed data}}\right) \times 100$$

$$\textit{Vertex Overhead} = \frac{\#\ \textit{of duplicated vertices}}{\#\ \textit{of vertices}} \times 100$$

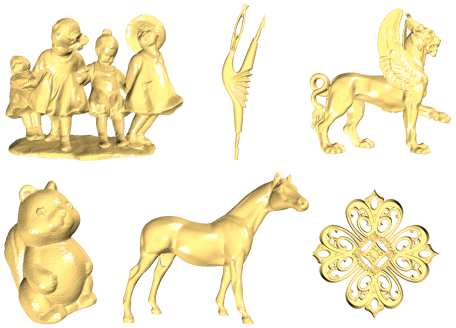


Fig. 5. Models used in our experiments. From the top left in the clockwise order, children, dancer, feline, filigree, horse, and squirrel.

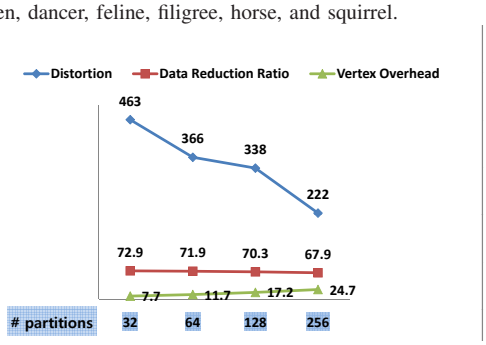


Fig. 6. Compression results with different numbers of partitions for dancer model. Distortions are shown in the unit of 10^{-6}.

The distortion of a mesh restored from encoded geometry is measured from the original mesh in the average $\mathcal{L}^2$ norm using the Metro tool [4].

Fig. 6 shows the distortions, data reduction ratios, and vertex overheads for increasing numbers of partitions with the dancer model. When the number of partitions becomes larger, the data reduction ratio decreases because we need to store more partition headers and the vertex overhead increases. In contrast, the distortion of the restored mesh is reduced with more partitions because the bounding cubes of partitions become smaller, which increases the accuracy of quantization. As we can expect, such tendency appeared for all models in our experiments.

Table I shows the distortions of restored meshes with different quantization methods. As shown in Fig. 6, the distortion from our local quantization varies with the number of partitions. In Table I, we use the number of partitions which provides 70% (+0.1) data reduction ratio. Table I shows that the distortion of 8-bit local quantization is quite less than that of 8-bit global quantization for every model. In addition, the restored quality of 8-bit local quantization is better than or similar to 11-bit global quantization in most cases, except the Squirrel model which is the smallest.

TABLE I
COMPARISON OF DISTORTIONS

Model	# V	# P	11-bit Global	8-bit Global	8-bit Local
Children	100,000	564	0.000436	0.003429	0.000395
Dancer	24,998	135	0.000449	0.003622	0.000293
Feline	49,864	270	0.000442	0.003583	0.000373
Filigree	514,300	2480	0.000452	0.003642	0.000160
Horse	19,851	128	0.000430	0.003375	0.000462
Squirrel	9,995	52	0.000416	0.003360	0.000966

(a) 8-bit Global (b) 8-bit Local

Fig. 7. Visual quality comparison between 8-bit global and our 8-bit local quantization.

Fig. 7 shows visual comparison of the restored meshes from global and local quantization with 8 bits. We can clearly see visual degradation in Fig. 7(a), while Fig. 7(b) is almost indistinguishable from the original in Fig. 5. This example demonstrates that 8-bit local quantization has enough accuracy for representing the vertex geometry.

By controlling the number of partitions for our 8-bit local quantization, we can achieve a similar level of distortions to global quantization with a specified number of bits. Table II shows experimental results with the dancer model. In Table II, we can see that the compressed data size for vertex geometry with local quantization is always smaller than the corresponding global quantization. Table II also implies that we can easily satisfy a requirement on the amount of distortions by controlling the number of partitions used for local quantization.

TABLE II
COMPARISON BETWEEN GLOBAL AND LOCAL QUANTIZATION

Global Quantization			Local Quantization		
b/v	$\mathcal{L}^2$ d-error	size (KB)	# partitions	$\mathcal{L}^2$ d-error	size (KB)
10	0.000884	92	10	0.000840	76
11	0.000449	101	40	0.000427	80
12	0.000228	110	256	0.000222	94

VII. SUMMARY AND DISCUSSION

The compression rate of our geometry encoding is $24 + \alpha$ bits per vertex (bpv), where α comes from the overhead of partition headers and duplicated vertices. This rate is far worse than the state-of-the-art techniques, which give about 10 to 13 bpv for geometry encoding [1], [7]. However, our goal was to develop a mesh geometry compression technique suitable for mobile graphics, instead of achieving best compression rates.

Our technique has the following nice properties;

- *simplicity* It is simple enough for real-time decoding on a mobile device. The decoding operation is combined with the model geometry transformation, and the only overhead is one matrix multiplication for each mesh partition.
- *on-the-fly decoding* At rendering time, vertex positions are restored on-the-fly through the model geometry transformation. Vertex data can be stored in main memory and transmitted to graphics hardware in the quantized form, saving memory space and system bus bandwidth.
- *compatibility* Since the decoding operation is compatible with the standard graphics pipeline, in addition to mobile graphics, our technique can be used for PC and game console graphics without modification.

Our geometry compression scheme can be combined with any data structure (e.g., triangle strips) used for representing meshes in main memory. In our scheme, a given mesh is partitioned into sub-meshes. When each sub-mesh is stored in main memory using the selected data structure, our local quantization method can be used for representing the vertex coordinates with reduced memory space. Similarly, our scheme can be adapted for a mesh file format. For example, for an OBJ file, we can set each sub-mesh as a group and represent vertex coordinates in the quantized form, where additional information, similar to the partition header in Fig. 4, is stored for each group.

ACKNOWLEDGMENTS

The dancer, dancing children, filigree, and feline models are courtesy of IMATI, IMATI-GE, SensAble technologies, and Multi-Res Modeling Group at Caltech, respectively. The dancer, dancing children, filigree, horse, and squirrel models are provided by the AIM@SHAPE Shape Repository.

REFERENCES

[1] P. Alliez and C. Gotsman, "Recent advances in compression of 3D meshes," in *Advances in Multiresolution for Geometric Modelling*, N. A. Dodgson, M. S. Floater, and M. A. Sabin, Eds. Springer-Verlag, 2005, pp. 3–26.

[2] D. Calver, "Vertex decompression in a shader," in *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, W. F. Engel, Ed. Wordware, 2002, pp. 172–187.

[3] S. Choe, M. Ahn, and S. Lee, "Feature sensitive out-of-core chartification of large polygonal meshes," in *Proc. of Computer Graphics International 2006*, 2006, pp. 518–529.

[4] P. Cignoni, C. Rocchini, and R. Scopigno, "Metro: Measuring error on simplified surfaces," *Computer Graphics Forum*, vol. 17, no. 2, pp. 167–174, 1998.

[5] M. Garland, A. Willmott, and P. S. Heckbert, "Hierarchical face clustering on polygonal surfaces," in *Proc. 2001 ACM Symposium on Interactive 3D Graphics*, 2001, pp. 49–58.

[6] S. Lloyd, "Least square quantization in pcm," *IEEE Transaction on Information Theory*, vol. 28, pp. 129–137, 1982.

[7] J. Peng, C.-S. Kim, and C.-C. J. Kuo, "Technologies for 3D mesh compression: a survey," *Journal of Visual Communication and Image Representation*, vol. 16, no. 6, pp. 688–733, 2005.

[8] K. Pulli, T. Aarnio, K. Roimela, and J. Vaarala, "Designing graphics programming interfaces for mobile devices," *IEEE Computer Graphics and Applications*, vol. 25, no. 8, pp. 66–75, 2005.

[9] B. Purnomo, J. Bilodeau, J. D. Cohen, and S. Kumar, "Hardware-compatible vertex compression using quantization and simplification," in *Proc. Graphics Hardware 2005*, 2005, pp. 53–61.

[10] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe, "Multi-chart geometry images," in *Proc. Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, 2003, pp. 146–155.

[11] J. Ström and T. Akenine-Möller, "iPACKMAN: High-quality, low-complexity texture compression for mobile phones," in *Proc. Graphics Hardware 2005*, 2005, pp. 63–70.