A Method for Storing Clustering Information of Model Simplification in GPUs

Takayuki Kanaya
Hiroshima International University[1)]
555-36, Kurose Gakuendai
Higashi Hiroshima, Hiroshima
Japan (739-2695)
t-kanaya@hw.hirokoku-u.ac.jp

Tomoaki Taniguchi
Osaka Institute of Technology[2)]
1-79-1, Kitayama
Hirakata, Osaka
Japan (573-0196)
taniguchi@ggl.is.oit.ac.jp

Yuji Teshima
Sasebo National College of Technology[3)]
1-1, Okishinchou
Sasebo, Nagasaki
Japan (857-1171)
teshima@post.cc.sasebo.ac.jp

Koji Nishio
Osaka Institute of Technology[2)]
nishio@is.oit.ac.jp

Kenichi Kobori
Osaka Institute of Technology[2)]
kobori@is.oit.ac.jp

ABSTRACT

A vertex-clustering simplification is a kind of model simplification. It is difficult for the vertex-clustering simplification to simplify complex models in real-time, although it is known as a very fast method. In addition, it is also difficult for the vertex-clustering simplification to control the number of faces. It synthesizes vertices in each cluster. Therefore, models sometimes consist of the unexpected number of faces. In recent years, Graphics Processing Units (GPUs) have grown so significantly in performance that both the computational speed and the computational accuracy improve spectacularly. GPUs have programmable units such as vertex shaders and geometry shaders. With shaders, GPUs can be used not only for graphics rendering but also for general purposes.

In this paper, we propose a real-time simplification algorithm for complex models of 3D objects by using a GPU whose performance gets better these days. First, vertex-clustering information is stored to video memory on a GPU. Next, the faces are reduced by the vertex-clustering information using a programmable shader, depending on the level of detail which a user defined. We also discuss a method to control the number of faces easily.

Keywords

Simplification, GPU, Vertex-Clustering, Real-Time rendering

1. INTRODUCTION

In recent years, it has been easy to express complex models of 3D objects in product design, simulation, medicine and games electronically, due to progressive computer technology and progressive computer graphics. However, it is still difficult to render them in real time. Therefore technology is required to change the level of detail (LOD) of multi-resolution representations of models according to a user's needs.

A vertex-clustering algorithm is known as a kind of model simplification. While the vertex-clustering algorithms feature low computational costs, they have 2 disadvantages. One is that model simplification has traditionally not been viewed as a real-time rendering on CPU, the other is that it is difficult for a user to control the degree of the LOD. The vertex-clustering algorithm has been proposed by Rossignac and Borrel [Rossignac and Borrel 1993]. This is the method where a cell, which includes all the vertices that exist in 3D space, is uniformly divided; all vertices within each grid cell are collapsed to a single representative vertex, which may be one of the input vertices or some weighted combination of the input vertices. Some triangles degenerate to edges or points. A number of other vertex clustering algorithms have also been proposed. Low and Tan have proposed a modified vertex-clustering algorithm using floating cells rather than a uniform grid [Low and Tan 1997]. The floating-cell clustering leads to more consistent simplification. Since the importance of vertices controls the positioning of clustering cells, the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

unpredictable simplification artifacts are greatly reduced. Luebke and Erikson have proposed the algorithm using a hierarchical grid in the form of an octree, which is called Tight Octree [Luebke and Erikson 1997]. This vertex tree allows for a dynamic, view-dependent adaptation of the level of detail. Kanaya and Kobori have proposed the improved Tight Octree algorithm allowing for cell size [Kanaya and Kobori 2002]. This method is easier to control the LOD level. Lindstrom has used QEM(quadric error metrics) [Garland 1999] to improve the positioning of the representative vertices [Lindstrom 2000]. Shaffer and Garland have proposed BSP-Tree partitioning to control the LOD level [Shaffer and Garland 2001], and have also used QEM to improve the positioning of the representative vertices [Garland and Shaffer 2002]. Mesh simplification has been a slow, CPU-limited operation performed as a pre-process on static meshes.

In recent years, GPUs have grown so significantly in performance that the computational speed and the computational accuracy improve spectacularly. Additionally, a general–purpose GPU, such as numerical calculation, modeling, have been studied by a Programmable Shader's appearance.

[DeCoro and Tatarchuk 2007] is famous as a model simplification using GPU. This method is based on [Lindstrom 2000] adopted to the novel GPU pipeline. Additionally, they have proposed the algorithm for variable level-of-detail, called probabilistic octree. This method is much faster than CPU-based algorithm.

We present an algorithm overcoming the 2 disadvantages of vertex-clustering algorithms. For this, we present a novel general-purpose data structure designed for a GPU. At the same time, we present a method to linearize a relationship between a level which a user defined and the number of faces. As a result, our algorithm is faster and the number of faces is easier to control.

2. Our algorithm

We outline our algorithm, which consists of the following five steps:

1. The space division algorithm.
2. Synthesis of vertices in each cluster.
3. Updating level for controlling the number of faces.
4. Storing the clustering information in a GPU.
5. Generating the simplified models.

The above procedures (1) to (2) are performed on the CPU; the procedures (3) to (5) are performed on the GPU.

2.1 Space division algorithm

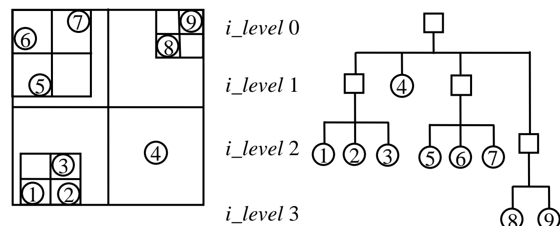
We applied the algorithm of [Kanaya and Kobori 2002], which we have proposed, to an arbitrary 3-D space division. The space division algorithm consists of the following five steps:

1. A cell, which includes all the vertices that exist in 3D space, is generated. The cell is a minimized cuboid, which is divided by domain parallel to the planes of the x-y coordinate, the y-z coordinate, and the z-x coordinate so that two or more vertices can be included.
2. The cell is equally divided by eight, and then eight cuboids are made.
3. The degree of the LOD (level of details) is determined by the longest edge of each cuboid. This is estimated by the following equation.

   $$i_level = \mathrm{INT}(\log_2 \frac{Length_{\max}}{Length_{current}}) \quad (1)$$

   where, INT() is an integral function. The $Length_{max}$ is the longest length of cell that envelops all vertices. The $Length_{current}$ is the longest length of the considered cell. The *i_level* is integer value.
4. The number of vertices belonging to each cuboid is equally divided into eight; dividing will end if the number of the vertices, which exists in the cuboid, is only one.
5. Otherwise, each remaining cuboid is changed into a cell and processing returns to step 2.

By this procedure, a tree can be generated as shown in Figure 1. This tree is an octree, since a cell is always divided by eight. Considering the degree of the LOD in this octree, the root node is an *i_level* 0 and the degree of the LOD becomes larger as the node is closer to a leaf node. The octree is clustering information.



(a) The space division. (b) An octree of (a).

Figure 1. A generated octree using a spatial partitioning. (In 2D space)

2.2 Synthesis of vertices in cluster

A method for the synthesizing of vertices in cluster is based on [Kanaya and Kobori 2002]. This is why we simply want to compare CPU-based algorithms with GPU-based algorithms. It is possible to use [Lindstrom 2000] instead of Kanaya and Kobori for accuracy improvement.

2.3 Updating level for controlling the number of faces

To make it easier for the user to define the number of faces, we propose a way of linearizing a relationship between a level which a user defined and the number of faces as follow.

First, we cluster the internal nodes which *i_level* is same. However, the longest length of cell varies according to cell in spite of the same *i_level*. The internal nodes are arranged in descending order of the length in each cluster which has the same *i_level*. Then, we call the level which has the rank in descending order in each cluster "*r_level*". The *r_level* is a real number. The *r_level* of each internal node m "r_level_m" is calculated by equation (2).

$$r_level_m = i_level_{before} + \frac{1}{number_i} \times k \quad (2)$$

where, i_level_{before} is the *i_level* of each internal node. k is a rank of each internal node after arranging in descending order. $number_i$ is the number of internal node in arbitrary *i_leveli*. The leaf level is not updated.

2.4 Storing the clustering information in a GPU

The clustering information which was generated by the above process is stored in a texture on the GPU as texture. We prepare 2 kinds of textures for storing the clustering information. One texture is called "Tree-buffer"; the other texture is called "Leaf-buffer". The Tree-buffer stores the information of the internal nodes in an octree. The Leaf-buffer stores the information of the leaf nodes. We illustrate the way to store the information of nodes in textures in Figure 2.

First, we describe a method used for storing the internal nodes in the Tree-buffer. Each texel of the Tree-buffer stores the synthesized coordinate of vertices and the r_level_m as shown in Figure 3, while arbitrary internal nodes with parent-child relationship arrange a column parallel to the v axis, as shown in Figure 2.

Next, we describe a method used for storing the leaf nodes in the Leaf-buffer. Each texel of the Leaf-buffer stores a coordinate of vertices of leaf node in the octree and a pointer whose value refers directly to

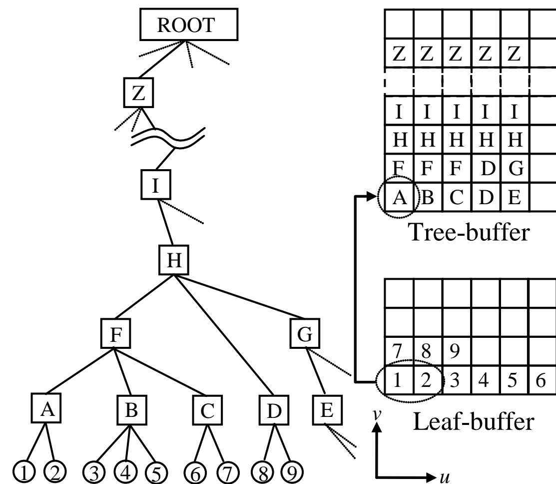


Figure 2. A stored state in the Tree-buffer and the Leaf-buffer.

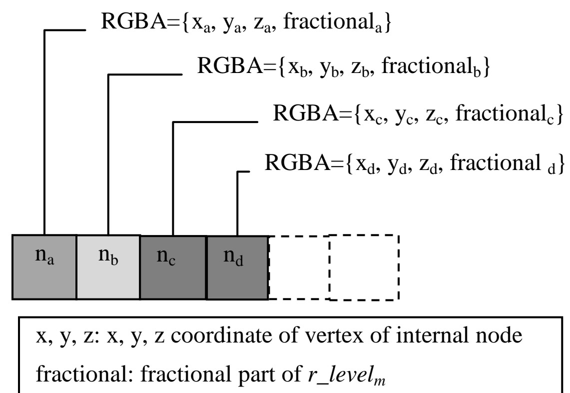


Figure 3. Method used for storing information in the Tree-buffer.

the parent node in the Tree-buffer as shown in Figure 4, in order from the bottom left as shown in Figure 2. For example, as we show in Figure 2, the leaf node 1, 2 hold a pointer whose value refers directly to the internal node A which is the parent for both nodes, respectively. As a result, it is easy to refer to a node

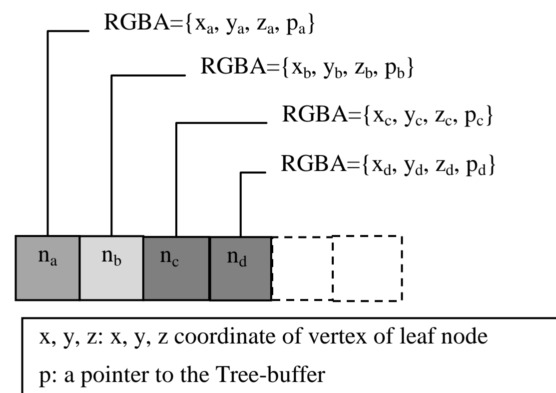


Figure 4. Method used for storing information in the Leaf-buffer.

of a level a user wants, when the texture coordinate of the terminal internal node is known.

2.5 Generating the simplified models

We describe the way for simplifying models by using the Tree-buffer and the Leaf-buffer on a GPU. First, we outline the generating procedure of the simplified model.

1. Calculate "*r_level*" of the number of faces "*f_count*" which a user wants.
2. Search the Leaf-buffer and the Tree-buffer for the coordinate of vertices associated with *r_level*.
3. Output each triangle consisted of the above coordinate of vertices.

(Pass 1) Calculate the level of detail of *f_count*.

We illustrate a way of calculating *r_level* of the number of faces a user wants in Figure 5. The *r_level* is the degree of the LOD, which is associated with a user-defined number of faces, in an octree. f_count_{max} in Figure 5 is the number of faces constructing the original model.

First, we search *i_level i* which is the maximum number of faces within f_count_i, *i_level i+1* which is the minimal number of faces over f_count_{i+1}. It is easy to set up the number of faces of the simplified model by counting these levels respectively in advance. Next, we calculate *r_level* corresponding to the *f_count*. In general, a model consists of nearly

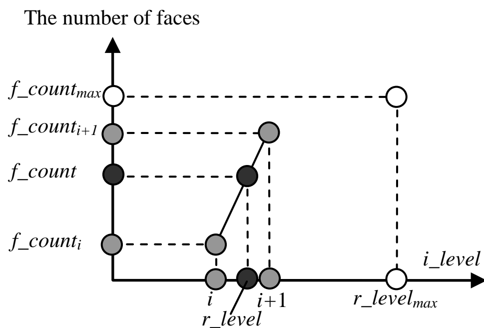


Figure 5. A computation of *r_level*.

uniform faces in 3D-CG. Therefore, we can think that the change in the number of faces is nearly constant, each time a vertex is deleted. We suggest the following equation for calculating the *r_level*.

$$r_level = i + \frac{f_count - f_count_i}{f_count_{i+1} - f_count_i} \quad (3)$$

We can generate the simplified models consisting of faces whose number is nearly equal to the user-defined number by using the *r_level*.

(Pass 2) Select coordinate of vertices.

Each coordinate of vertices is selected by *r_level* on a vertex shader of a GPU. We illustrate the selecting way with an example in Figure 6.

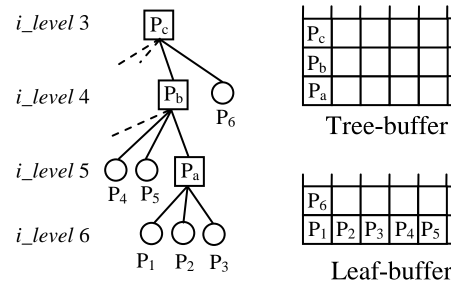


Figure 6. An example of a texture in an octree.

We describe how to calculate a texture coordinate of a parent node associated with a leaf node on the Tree-buffer, when we assume *r_level* is equal to "4.4".

First, an arbitrary leaf node is selected. In this case, we assume that P_1 is selected. When the pixel of P_1 in the Leaf-Buffer is referred, it stored a pointer whose value refers directly to the parent node P_a according to Table 1. We can calculate the texture coordinate of P_a in the Tree-buffer by the pointer.

Table 1. Table of referenced nodes in each leaf node.

Leaf node	$level_m$
P_1	P_a (5.633)
P_2	
P_3	
P_4	P_b (4.382)
P_5	
P_6	P_c (3.457)

Next, we calculate a node associated with *r_level* by nodes whose parent node is located on the same axis of *v*. As described above, all parent nodes which are able to be traced from arbitrary internal nodes are arranged a column of the Tree-buffer texture. Therefore, we calculate the location storing the target node with difference *d* between *r_level* and $level_l$.

$$d = level_l - r_level \quad (4)$$

where, $level_l$ is level of leaf node.

However, the target node can not always be calculated by equation (4), because both *r_level* and level which internal nodes have are real numbers. Therefore, we consider the following 3 cases according to *d*. These are (Case 1) $d \le 0$, (Case 2) $0 < d \le 1.0$, (Case 3) $d > 1.0$. In (Case 1), the three world coordinates of an arbitrary vertex in the

leaf node are selected, because the level of the leaf node is too small for the *r_level*. In (Case 2), if a level of the parent node for the leaf node is larger than the *r_level*, the three world coordinates of an arbitrary vertex of the leaf node are selected; otherwise, the three world coordinates of an arbitrary vertex of the parent node are selected. In (Case 3), when each level of 2 internal nodes closest to the *r_level* is compared with the *r_level*, the three world coordinates of an arbitrary vertex of the internal node whose level is larger than the *r_level* are selected. In Figure 4, $level_l$ is 6, *r_level* is 4.4. *d* is equal to 1.6 by equation (4). As a result, this is selected as Case 3. The candidate internal nodes are P_a and P_b. The selected node is P_a because *r_level* (4.4) is greater than the level of P_b (4.382).

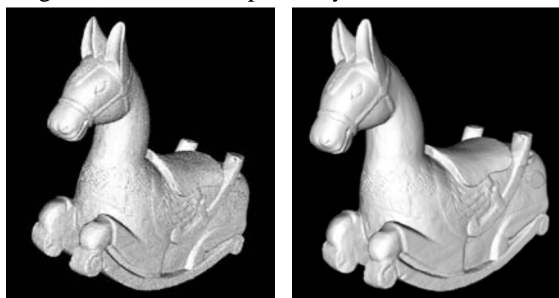
As described above, the access number of times to texture is 3 tops. It is quicker to search in the octree.

(Pass 3) Generate triangles for meshes.

The above process selects each coordinate of vertices for triangles. This pass determines if each triangle is generated in geometry shader. The geometry shader calculates normal vectors of triangles. If the size of the normal vector is not equal to 0, triangle is rendered; otherwise, triangle is not rendered.

3. Experiment and Results

To evaluate the performance of our algorithm, we made 3 experiments. The first experiment is associated with processing time, the second is associated with controlling the number of faces. Figure 7 shows 2 original models and the simplified models whose number of faces are one tenth of original models, respectively. We use "*s_level*" instead of the number of faces, where *s_level* is degree of LOD. The range of *s_level* a user defines is from 1 to 1,000. That is, when *s_level* is 1,000, the number of faces is that constructing the original model. When *s_level* is 1, the simplified model consists of one thousandth the number of faces for the original model. All simplifications were performed on a PC with a Core2 Duo CPU (2.66GHz), 2GB of RAM and an NVIDIA GeForce 8800GTX (768MB) GPU, collected on Windows XP Professional.



(a) Model A (2,208,936) (b) The simplified model of A (226,866)

(c) Model B (3,745,150) (d) The simplified model of B (377,278)

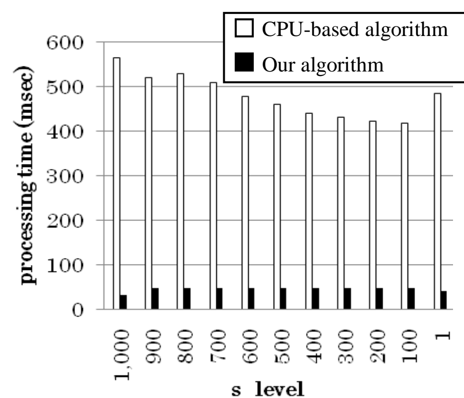
Figure 7. Experimental models and Simplified results.

3.1 Experiment associated with processing time.

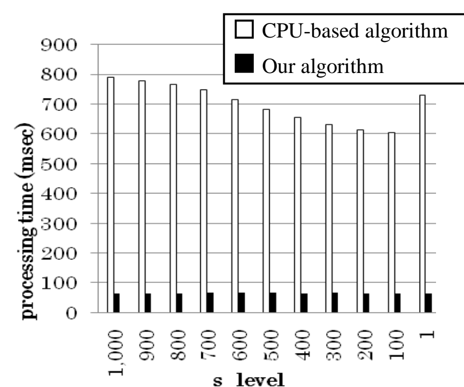
We compare our algorithm with a CPU-based algorithm. The CPU-based algorithm is [Kanaya and Kobori 2002]. Figure 8 shows the comparison of 2 algorithms. The horizontal axis shows the *s_level* and the vertical axis shows processing time. Our algorithm is up to 17 times quicker than the CPU-based algorithm, as shown in Figure 8. Additionally, it is found that the processing time is almost constant at every *s_level*.

3.2 Experiment associated with controlling the number of faces

We verify that when a user defines an *s_level*, the simplified models consist of the number of faces he/she expected. The range of *s_level* a user defines



(a) Processing Time of Model A



(b) Processing Time of Model B

Figure 8. Comparisons of our algorithm and CPU-based algorithm.

is from 1 to 1,000. That is, when *s_level* is 1,000, the number of faces is that constructing the original model. When *s_level* is 1, the simplified model consists of one thousandth the number of faces for the original model. Figure 9 shows a result of model A simplified by using our algorithm. The horizontal axis shows the *s_level* and the vertical axis shows the number of faces.

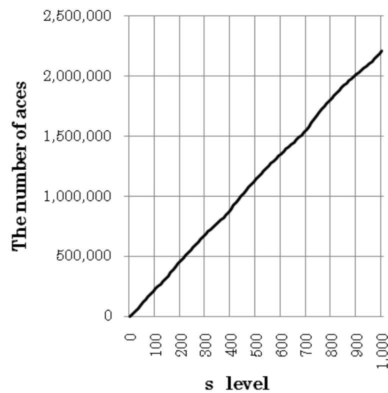


Figure 9. Relationship between *s_level* and the number of faces.

A relationship between *s_level* and the number of faces is almost linearized according to Figure 9. However, the difference between the numbers of faces we have expected and actual results were accurate within 4.7%, because we calculated *r_level* of *s_level* by linear approximation. However, it is easy to get the number of faces a user wants by using our method.

4. Conclusions

We have presented a method for model simplification on the GPU programmable shader and demonstrated how triangle decimation becomes practical for real-time use. We have applied a vertex-clustering algorithm to the GPU and we have presented how to store the clustering information.

We introduced "*r_level*" that is a real number, for the purpose of generating the simplified models which consisted of the number of faces a user expected. We introduced 2 buffers "Tree-buffer" and "Leaf-buffer" for faster access and efficient storage. Additionally, the experiment results showed efficacy.

We have not compared our algorithm with the algorithm based GPU [DeCoro and Tatarchuk 2007]. We would like to compare our algorithm with the algorithm in the future work.

5. REFERENCES

[DeCoro and Tatarchuk 2007] DeCoro, D. and Tatarchuk, N., Real-time Mesh Simplification Using the GPU. Symposium on Interactive 3D Graphics (I3D) 2007, pp. 6, April 2007.

[Garland and Shaffer 2002] Garland, M., and Shaffer, E., A multiphase approach to efficient surface simplification. In VIS '02: Proceedings of the conference on Visualization '02, IEEE Computer Society, Washington, DC, USA, 117–124, 2002.

[Garland 1999] Garland, M., Quadric-based polygonal surface simplification. PhD thesis, Carnegie Mellon University. Chair-Paul Heckbert, 1999.

[Kanaya and Kobori 2002] Kanaya, T., and Kobori, K., A Method of Model Simplification Using Spatial Partitioning. In the journal of the Institute of Image Information and Television Engineers 56(4), 636-642, 2002.

[Lindstrom 2000] Lindstrom, P., Out-of-core simplification of large polygonal models. In SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 259–262, 2000.

[Low and Tan 1997] Low, K.-L., and Tan, T.-S., Model simplification using vertex-clustering. In SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics, ACM Press, New York, NY, USA, 75–82, 1997.

[Luebke and Erikson 1997] Luebke, D., and Erikson, C., View-dependent simplification of arbitrary polygonal environments. In SIGGRAPH ' 97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques, ACM PRESS/Addison-Wesley Publishing Co., New York, NY, USA, 199–208, 1997.

[Rossignac and Borrel 1993] Rossignac, J., and Borrel, P., Multi-resolution 3d approximations for rendering complex scenes. In Geometric Modeling in Computer Graphics, Springer-Verlag, New York, NY, USA, 455–465, 1993.

[Shaffer and Garland 2001] Shaffer, E., and Garland, M., Efficient adaptive simplification of massive meshes. In VIS '01: Proceedings of the conference on Visualization '01, IEEE Computer Society, Washington, DC, USA, 127–134, 2001.