Theoretical and Applied Informatics
ISSN 1896–5334
Vol.23 (2011), no. 3-4
pp. 263–275
DOI: 10.2478/v10179-011-0018-4

Optimization of Mesh Representation for Progressive Transmission

KRZYSZTOF SKABEK, DARIUSZ POJDA

Institute of Theoretical and Applied Informatics
of the Polish Academy of Sciences

Received 25 October 2011, Revised 4 December 2011, Accepted 5 December 2011

Abstract: The paper presents the implementation of methods for progressive mesh encoding. The described implementation is a modification of the software published in 2008 by Zhao He, which is based on studies by Michael Garland and Hugues Hoppe.

We focused on improving the performance of the software. In particular we modified the data structures to ensure their better indexing, which significantly improved the computational complexity of the algorithm. We also implemented the support for simplified meshes with textures.

The authors present the comparison of the performance of these methods in relation to the original He Zhao's algorithm. The performance of progressive mesh encoding in our implementation was significantly improved.

Keywords: progressive mesh, quadric error, mesh transmission

1. Introduction

There are some ways to make the processing, transmission and presentation of 3D complex mesh objects more efficient. One of the improvements defines several levels of details for the mesh object, e.g. to display the more detailed model when viewer is coming closer. Transmitting a mesh over communication line one may want to see a model with a coarse shape approximation and next increase levels-of-details approximations. Mesh storing is very memory consuming. Such problem may be solved in different ways and one of them is preparing progressive meshes for both mesh simplification or compression.

There are many different ways to represent graphical 3D models, in this article we focused on *progressive meshes* which were introduced by Hoppe [5]. We assume here that the progressive mesh complexity does not depend on viewer position, it is simplified on the whole surface.

2. Progressive Representations – Overview

Many techniques have been proposed to compress and transmit mesh data. They can be divided into nonprogressive and progressive methods. The first group comprises methods which encode the entire data as a whole. They can either use the interlocking trees (vertex spanning tree and triangle spanning tree) or utilize the breadth-first traversial method to compress meshes. On the other hand there are methods that perform mesh compressing progressively. The solution proposed by Hoppe [5] enables continuous transition from the coarsest to the finest resolution. In such case a hierarchy of level-of-detailed approximation is built. Also the efficient quadric algorithm for mesh decimation was proposed by Hoppe [4] and further extended by Garland [3].

In the article [12] another view-dependent graphics streaming scheme was proposed by Yang, Kin and Kuo. 3D models are split into several partitions and they are simplified and coded separately. The compressed data is sent on the user request. The partitioning of the model is done arbitrary and the separated partitions are simplified by merging inner vertices into a single vertex.

2.1. Mesh Representation for Progressive Transmission

The mesh geometry can be denoted by a tuple (K, V) [4], where K is *a simplicial complex* specifying the connectivity of the mesh simplices (the adjacency of the vertices, edges, and faces), and $V = \{v_1, \ldots, v_m\}$ is the set of vertex positions defining the shape of the mesh in R^3. More precisely, we construct a parametric domain $|K| \subset R^m$ by identifying each vertex of K with a canonical basis vector of R^m, and define the mesh as the image $\phi_v(|K|)$ where $\phi_v : R^m \rightarrow R^3$ is a linear map.

Besides the geometric positions and topology of its vertices, the mesh structure has another appearance attributes used to render its surface. These attributes can be associated with faces of the mesh. A common attribute of this type is the material identifier which determines the shader function used in rendering a face of the mesh. Many attributes are often associated with a mesh, including diffuse colour (r, g, b), normal (n_x, n_y, n_z) and texture coordinates (u, v). These attributes specify the local parameters of shader functions defined on the mesh faces. They are associated with vertices of the mesh.

We can further express a mesh as a tuple $M = (K, V, D, S)$, where V specifies its geometry, D is the set of discrete attributes d_f associated with the faces $f = \{j, k, l\} \in K$, and S is the set of scalar attributes $s(v, f)$ associated with the corners (v, f) of K.

As many vertices may be connected in one corner with the same attributes, the intermediate representation called *wedge* was introduced to save the memory [6]. Each vertex of the mesh is partitioned into a set of one or more wedges, and each wedge contains one or more face corners. Finally we can define the mesh structure that contains an

array of vertices, an array of wedges, and an array of faces, where faces refer to wedges, and wedges refer to vertices. Face contains indices to vertices, additionally this structure contains array of face neighbours (*fnei*) in which indices of tree adjacent faces are stored, this information is necessary to build a progressive mesh. There is nothing said in reference papers about order of vertices and indexes of adjacent faces in face structure. In our implementation the counter is stored clockwise and additionally first adjacent face is at first position as first vertex, so if we cross first edge we find the first neighbour, if we cross second we find the second, etc.

In many places of this article we use the word *edge*. The edge is a connected pair of vertices or, in other words, it is a pair of adjacent vertices. There is no additional list of edges, but the first vertex and the face to which this edge belongs are defined instead. Using wedge we can access vertex, even if the adjacent face does not exist we can define edge. Definition of edges is necessary to simplify meshes, to create progressive meshes as well as to determine which edge (vertex) could be collapsed.

2.2. Construction of Progressive Meshes

Progressive mesh (PM) [5] is special case of a mesh or rather an extension of mesh representation, it makes it possible to build mesh for different level-of-details (LOD) [8]. It also allows loading the base mesh M^0, as the mesh of the lowest LOD, and then process the loading of the remaining parts of the mesh structure. As an input source we may use a memory input stream.

In PM form, an arbitrary mesh $\widehat{M}$ is stored as a much coarser mesh M^0 together with a sequence of n detail records that indicate how to incrementally refine M^0 exactly back into the original mesh $\widehat{M} = M^n$. Each of these records stores the information about *a vertex split*, an elementary mesh transformation that adds an additional vertex to the mesh. Thus the PM representation of $\widehat{M}$ defines a continuous sequence of meshes $M^0, M^1, \ldots M^n$ of increasing accuracy, from which LOD approximations of any desired complexity can be efficiently retrieved. Moreover, smooth visual transitions (*geomorphs*) [5] can be efficiently constructed between any two such meshes. In short, progressive meshes offer an efficient, lossless, continuous-resolution representation. Progressive meshes makes it possible not only to store the geometry of the mesh surface, but, what is more important, preserve its overall appearance, as defined by the discrete and scalar attributes associated with the surface.

There are three operations that make it possible to determine the base mesh of $\widehat{M}$: *edge collapse*, *vertex split* and *edge swap*. Edge collapse operation is sufficient to successfully simplified meshes. Edge collapse operation $ecol(v_s, v_t)$ remove one edge and instead two vertices v_s and v_t insert new one v_s. Additionally two faces (v_t, v_s, v_l) and (v_t, v_r, v_s) are removed. The initial mesh M_0 can be obtained by applying a sequence of n edge collapse operations to $\widehat{M} = M^n$:

$$(\widehat{M} = M^n) \overset{ecol_{n-1}}{\rightarrow} \cdots \overset{ecol_1}{\rightarrow} M^1 \overset{ecol_0}{\rightarrow} M^0$$

Edge collapse operation is invertible. The inverse transformation is called vertex split. Vertex split operation adds in place of vertex v_s two new vertices v_s and v_t and two new faces (v_t, v_s, v_l), (v_t, v_r, v_s) if edge $\{v_s, v_t\}$ is boundary then adds only one face. Because edge collapse transformation is invertible our mesh $\widehat{M}$ can be presented as a simple M^0 and sequence of n vsplits records:

$$M^0 \overset{vsplit_0}{\rightarrow} M^1 \overset{vsplit_1}{\rightarrow} \cdots \overset{vsplit_{n-1}}{\rightarrow} (\widehat{M} = M^n)$$

We call $(M^0, vsplit_0, ..., vsplit_{n-1})$ a progressive mesh (PM) representation of M.

2.3. Quadratic-Based Mesh Reduction

In order to perform the mesh reduction it is necessary to select a sequence of edges to be removed. The problem of the proper choice may be solved in several ways. One of the most efficient methods is based on quadratic eror metrics [3].

Quadric Q is a symmetric matrix of size 4×4 that holds information about planes of neitghbour faces and can be defined as follows:

$$Q_v = \sum_{p_v} K_p$$

where: p_v is the set of planes containing faces and directly adjacent to the considered vertex v, K_p is the base error quadric stored also in matrix of size 4×4 and used to calculating the square distance from the plane of any point p.

Quadric matrix K_p can be formalized as:

$$K_p = pp^T = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix} \text{, where } p = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

For each vertex $v = \begin{bmatrix} v_x & v_y & v_z & 1 \end{bmatrix}^T$ the matrix Q is assigned and the quadratic form $\Delta v = v^T Q v$ calculated. This form is in fact the measure of error.

When removing the edge, points v_1 and v_2 are replaced by v_0 and the new matrix is assigned to v_0 computed as the sum of the matrices associated with the vertices v_1 and v_2: $Q_0 = Q_1 + Q_2$.

Quadratic form Δv is practically the sum of the square distances of vertex v to the planes containing the faces adjacent to v.

3. Implementation

Our goal was to implement an efficient method for building and transmission of progressive meshes. Another assumption was to enrich the data of spatial geometry with texturees. The initial implementation was done using Hoppe methods of progressive meshes [11], but finally we decided to extend the code developed by He Zhao in 2008 [13].

These implementations were chosen as a base of our tests because of good quality of the resultant progressive meshes and the assumed lossless mesh reduction which always makes it possible to fully reconstruct the original mesh. Another opportunity of such approach was the clearity of structures and open code easy to modifications.

He Zhao implemented the method for surface simplification using quadric error metrics provided by Garland [1], [3] and also Hoppe methods for progressive meshes [5], [6]. The implementation was published i n C++ under GNU General Public License.

The clue of the approach is selection of vertices for reduction and calculation of the new vertex. For each pair of vertices that are connected with edge, or optionally, for point cloud, that are closer than a given threshold, a quadric error is calculated. The quadric error is a measure of mesh distortion in case of reduction of vertices.

We modified the data structures to ensure the better indexing. In this way we significantly improved the computational complexity of the algorithm. We also implemented the support for simplified meshes with textures.

3.1. Performance Improvements

Quadric algorithm proposed by He Zhao appeared to be not sufficient in our application. While quality of meshes obtained by reduction was fully acceptable, the time needed for processing large meshes was a critical factor. Reduction of the mesh containing approximately 1.5 million vertices and 3 million faces took more than 200 hours, according to our tests as described in section 4. Such time consumption was unacceptable so we modified methods proposed by He Zhao.

We did not want to change the basic assumptions of the proposed algorithm, but in the first step of optinization we rewrote certain parts of code, which were characterized by the highest computational complexity and caused the inefficiency in the algorithm.

The principles of algorithm proposed by He Zhao are as follow:

1. for each pair of connected vertices calculate quadric error
2. while number of faces is greater then expected, do:
   (a) find the pair of vertices with smallest quadric error,
   (b) calculate coordinates for new vertex,

Fig. 1. Textured mesh waza.obj (134227 faces). Original (left) and decimated (right)

(c) find all faces containig any vertices belonging to the selected pair, and:

- remove all faces connected to both vertices,
- for all faces containing only one vertex found for pair, change it to the new vertex,

(d) recalculate quadric error for all pairs of vertices containing the new vertex.

We noticed that the most important problem was the nested loop. Note that in the software of He Zao points 2a and 2c were implenented as ordinary searching loop in the array. It can therefore be assumed that in the worst case the number of runs of the main loop is equal to $f_r * (p + f)$, where f_r is the number of faces to be reduced, p is the number of pairs of connected vertices, and f is the total number of faces in the mesh. Because of f_r can be equal to f, and p can be much greater than f, we can say, that computational complexity of this algorithm is $O(f^2)$.

We used tree structures for indexing arrays, and this way strongly reduced time of search data. We created two structures for indexing the error arrays. One of them was organized by quadric error values and the second was ordered by the vertex number. For faces array there was only one indexing structure, organized by the vertex number. With such solution, the time of execution of main loop was shortened and equal to $f_r * (log(p) + log(f))$. Thus the overall complexity of the algorithm is $O(f * log(f))$.

Additional time is required for the preparation of these tabular structures as well as for the modification of indices when data changes in main arrays. However, these overheads are negligible against the gained profits.

3.2. Support for Textured Meshes

The support for meshes with texture was implemented. The texturing method is based on a simple texture mapping, where the structure of texture, its color and brightness are not analyzed.

Fig. 2. Comparison of texture representation quality. Original (top left), reduction ratio = 0.5 (top right), reduction ratio = 0.1 (bottom left) and reduction ratio = 0.05 (bottom right)

We decided for the solution that does not change the texture coordinates (coordinates of points in the texture image) associated to the reduced vertices of the mesh. When we reduce a pair of vertices, we arbitrarily choose one of them and rewrite all references to the texture image associated with this vertex, to the newly created vertex. Such solution may cause some distortions of the resultant texturing. There is a shift in the texture on the mesh surface and a loss of some part of the projected texture image and thus the quality of the texture representation can be reduced. Such mapping distortions are getting bigger in each step of mesh reduction. We demonstrate the sample results of such effect in figures 1 and 2.

A quality of representation of texture on mesh which was coded with our method is worse than described in publications by Hoppe [7] and Garland [2]. This especially applies to strongly reduced meshes. Our next works will concern to improve the quality of the reducing and mapping of these textures.

3.3. Data Format for Fast Mesh Reconstruction

The output format of vsplit data generated by He Zhao's program is very simple. It contains coordinates of pairs of vertices reduced in each step, and coordinates of vertices replacing this pair. It is very good notation because of size of generated file and size of data needed for transmission of progressive mesh. Unfortunately, we have to make a lot of additional operations if we want to reconstruct original mesh from progressive form. At each step of reconstruction we need to search for vertex to be splited. This algorithm runs slowly when the simple vertex arrays are used. Moreover, we have not enough evidence about changes that we have to enter for faces containing the splitted vertices. We still do not know which of these faces should be connected and to which of the newly introduced vertices. It is possible to arbitrarily decide, of course, but then there is no certainty if the reconstructed mesh will be the same as original mesh before reduction to the progressive form.

To make the reconstruction process faster we gathered in the storage file all the data neccessary for direct reconstruction of the oryginal mesh. The format of vsplit data is shown below (values between '[' and ']' brackets occur conditionally when a preceding value is non-zero number):

s	$v_1\ x_1\ y_1\ z_1$
	$v_2\ x_2\ y_2\ z_2$
	N_{Fa} [$a\ b\ c$ ...]
	N_{Fc} [$a\ b\ c$ ...]
	N_{Ta} [$a\ b\ c$ ...]
	N_{Ca} [$i\ s\ t$ ...]
	N_{Cc} [$i\ s\ t$ ...]

Initial **s** is a char that sign a line with vsplit data. v_1 is an index of vertex that we need to split in this step. First of new vertices will have got the index v_1, and coordinates (x_1, y_1, z_1). Second of these vertices will have got index v_2, and coordinates (x_2, y_2, z_2). N_{Fa} is a number of faces we should add to mesh. If N_{Fa} is greater than zero, there are idices of vertices ($a\ b\ c$) for each of N_{Fa} added faces. Analogously, N_{Fc} means a number of faces that we need to change at this step. Index v_1 should to be changed to v_2 for all these faces. Three next sections are for textured meshes. N_{Ta} is a number of texture triangles that we need to add, and this value is usualy equal to N_{Fa}. N_{Ca} is a number of pairs of (s,t) coordinates on texture image that we must add, and similarly

N_{Cc} is a number of pairs that exists and have to be changed. Value i is an index, and values s and t are the pair of new coordinates assigned to the index i.

a) b) c) d)

Fig. 3. Comparison of quality for original (left) and decimated (right) meshes: a) cow.obj, b) czajnik.obj, c) abel.obj, d) sabinki.obj, e) krolowa.obj

4. Comparison

We performed simple tests for comparison of our implementation with He Zhao's code. We have decimated (reduced with factor 0.1) some meshes with varying size. Personal Computer with Intel Core 2 Quad Q6600 working at 2.4GHz, and with 8GB RAM was used for this test. All tests were carried out under Microsoft Windows XP Professional.

We selected five meshes of different sizes. Figure 3 shows them. Four of these meshes we took from the database of 3D mesh models of cultural heritage created under previously realized research project N N516 1862 33.

mesh name	number of faces	He Zhao's	our code
cow.obj	5804	1.4s	<1s
czajnik.obj	107890	869s	8s
abel.obj	203202	3277s	14s
sabinki.obj	1348030	181562s	102s
krolowa.obj	2999990	860400s	480s

Table 1: Comparison of times of mesh decimation achieved by the He Zhao's implementation and our code, depending on the original mesh size

The results of tests are explained in Table 1. We can say, that the performance of reduction for large meshes consisting of more then 200,000 faces was totally unacceptable in He Zhao's implementation. Reduction time of these meshes performed with our implementation was significantly better. All results was also shown in Figure 4.

The graph of the relationships between decimation time and mesh size for our implementation and He Zhao's code we have explained in Figure 5. The linear graph is

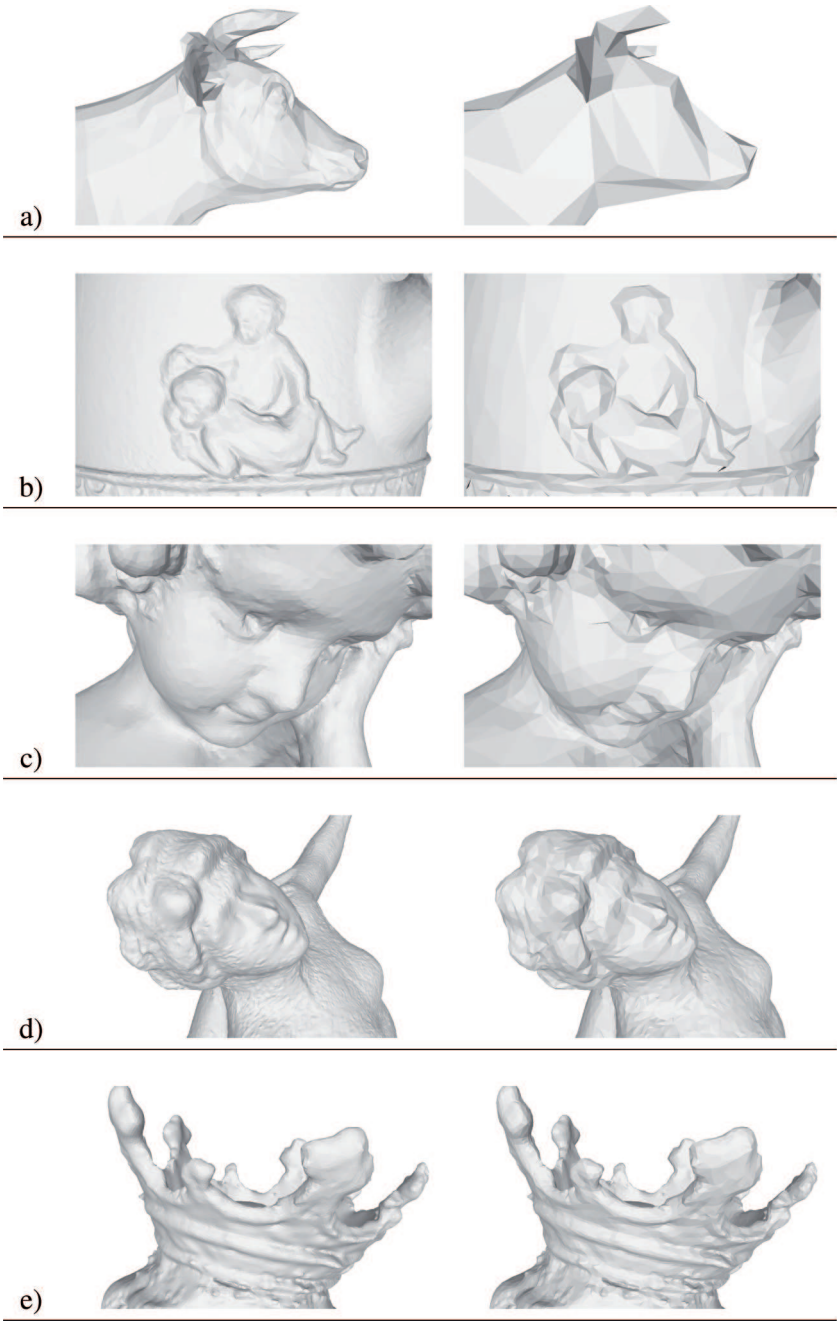
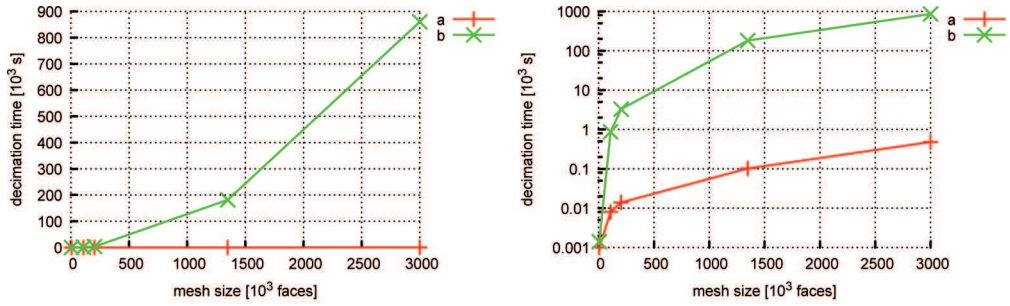


Fig. 4. Meshes used for tests: a) abel.obj, b) czajnik.obj, b) sabinki.obj, b) cow.obj



a) b)

Fig. 5. Linear (left) and logarithmic (right) graph of the relationship between decimation time and mesh size for our implementation (a) and He Zhao's code (b)

unclear, and suggests constant or linear complexity of our implementation. Of course this is not true, as we have shown at section 3.1. It can be seen on the more precise logarithmic graph.

5. Summary

We have implemented the algorithm for progressive codding of meshes based on Michael Garland's quadric error method. This is the modification of software provided by He Zhao in 2008.

As we have shown in tests, our implementation is significantly faster then He Zhao's code, and is suitable for use in the reduction of large meshes. We also have implemented support for encoding meshes with texture, and we have defined a new format of data files for storing of progressive meshes.

In further work we will improve the quality of the texture coding. Will be considered the characteristics of texture, as the distribution of color and brightness. Moreover we can see possibilities to further accelerate of the program. The software requires fine-tuning in terms of optimizing memory usage, too.

We have also developed the specification of the new file format based on XML language. This format is more readable and easier to process. It will be implemented in future versions of the software.

Acknowledgements. This work was supported in part by the Ministry of Science and Higher Education of Polish Government, under the research projects N N516 482340 and N N516 479740.

References

1. M. Garland, P. Heckbert: Surface Simplification Using Quadric Error Metrics. Computer Graphics (SIGGRAPH '97 Proceedings), 1997.
2. M. Garland, P. Heckbert: Simplifying Surfaces with Color and Texture using Quadric Error Metrics. IEEE Visualization 1998.
3. M. Garland: Quadric-Based Polygonal Surface Simpling. Pittsburgh : School of Computer Science Carnegie Mellon University, 1999.
4. H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle: Mesh Optimization. Computer Graphics (SIGGRAPH '93 Proceedings), pp. 19-26, 1993.
5. H. Hoppe: Progressive Meshes. Computer Graphics (SIGGRAPH '96 Proceedings), pp. 99-108, 1996.
6. H. Hoppe: Efficient Implementation of Progressive Meshes. Computer & Graphics, vol. 22(1), pp. 27-36, 1998.
7. P. Sander, J. Snyder, S. Gortler, H. Hoppe: Texture mapping progressive meshes. ACM SIGGRAPH 2001 Proceedings, pp. 409-416, 2001.
8. D. Luebke, M. Reddy, J.D. Cohen, A. Varshney, B. Watson, R. Huebner: Level of Details for 3D Graphics. Morgan Kaufmann, 2003.
9. F. Nielsen: Visual Computing: Geometry, Graphics and Vision. Charles River Media, 2005.
10. D. Luebke, M. Reddy, J.D. Cohen, A. Varshney, B. Watson, R. Huebner: Level of Details for 3D Graphics. Morgan Kaufmann Publishers, 2003.
11. K. Skabek, Ł. Ząbik: Implementation of Progressive Meshes for Hierarchical Representation of Cultural Artifacts, Compter Vision & Graphics, LNCS 5337, 2009.
12. S. Yang, C.-S. Kin, J. Kuo: A Progressive view Dependent Technique for Interactive 3-D Mesh Transmission, IEEE Transaction on Circuitsfor Video Technology, vol. 14, no. 11, pp. 1249-1264, 2004.
13. He Zhao: A surface simplification software. WWW Site: http://hezhao.net/projects/progressive-meshes/, 2008.

Optymalizacja reprezentacji siatek progresywnych dla potrzeb przesyłu danych

Streszczenie

W artykule przedstawiono implementację algorytmu kodowania siatek trójwymiarowych do postaci progresywnej. Opisana implementacja jest rozwinięciem oprogramowania opublikowanego w 2008 roku przez He Zhao, które bazuje na opracowaniach Michaela Garlanda [1] oraz Huguesa Hoppe [5].

Celem autorów było wykorzystanie prezentowanego oprogramowania do kodowania siatek o znacznym rozmiarze, reprezentujących między innymi skany eksponatów muzealnych lub modele uzyskane w wyniku obrazowania medycznego. W obu przypadkach kodowanie do postaci progresywnej ma na celu zapewnienie efektywnego przesyłania i szybkiej prezentacji siatek na różnych poziomach wizualizacji szczegółów (levels of detals). Jednocześnie powinno ono umożliwiać szybkie odzyskanie oryginalnej siatki w razie potrzeby wykonania pomiarów.

Skupiono się na poprawie szybkości działania algorytmu poprzez wprowadzenie struktur danych zapewniających ich odpowiednie indeksowanie, co znacząco poprawiło złożoność obliczeniową. W porównaniu do możliwości oferowanych przez oprogramowanie He Zhao została także dodana uproszczona obsługa progresywnego kodowania siatek z teksturami.

Przeprowadzono i przedstawiono w artykule porównanie czasów działania oprogramowania w prezentowanej wersji oraz w implementacji He Zhao i wykazano, że wydajność czasowa uległa znaczącej poprawie.