Sliding-Tris: A Sliding Window Level-of-Detail Scheme

Oscar Ripolles, Francisco Ramos, and Miguel Chover

Universitat Jaume I, Castellon, Spain
{oripolle,jromero,chover}@uji.es

Abstract. Virtual environments for interactive applications demand highly realistic scenarios, which tend to be large and densely populated with very detailed meshes. Despite the outstanding evolution of graphics hardware, current GPUs are still not capable of managing these vast amounts of geometry. A solution to overcome this problem is the use of level-of-detail techniques, which recently have been oriented towards the exploitation of GPUs. Nevertheless, although some solutions present very good results, they are usually based on complex data structures and algorithms. We thus propose a new multiresolution model based on triangles which is simple and efficient. The main idea is to modify the list of vertices when changing to a new level of detail, in contrast to previous models which modify the index list, which simplifies the extraction process. This feature also provides a perfect framework for adapting the algorithm to work completely on the GPU.

Keywords: Multiresolution, Level of Detail, GPU, Sliding-Window.

1 Introduction

Nowadays, applications such as computer games, virtual reality or scientific simulations are increasing the detail of their environments with the aim of offering more realism. This objective usually involves dealing with larger environments containing lots of objects which amount to a large quantity of triangles. However, despite the constant improvements in performance and capabilities of GPUs, it is still difficult to render such complex scenes as vertex throughput and memory bandwidth become considerable bottlenecks when dealing with them. As a result, these environments cannot be interactively rendered by brute force methods.

Among the solutions to overcome this limitation, one of the most widely used is multiresolution modeling. A level-of-detail or multiresolution model is a compact description of multiple representations of a single object [1] that must be capable of extracting the appropriate representation in different contexts.

In recent years, many solutions based on level-of-detail techniques have been presented. Nevertheless, only a few exploit, in one way or another, some of the current GPU functionalities. However, although they provide interactive rates, they require complex data structures and algorithms to manage them. In this paper we describe a multiresolution model for real-time rendering of arbitrary meshes which contributes to diminish the existing distance between a multiresolution GPU-based solution and its implementation in any 3D application. Our approach includes the following contributions:

– A simple data structure based on vertex hierarchies adapted to the GPU architecture. The vertex hierarchy is given through the edge contraction operations of the simplification process [2].
– Storage cost with low memory requirements.
– Representations are stored and processed entirely in the GPU avoiding the typical bottleneck between the CPU and the GPU and thus obtaining a great performance by exploiting the implicit parallelism existing in current GPUs.

This paper presents the following structure. Section 2 contains a study of the work previously carried out on GPU-friendly multiresolution modeling. Section 3 presents the basic framework of Sliding-Tris. Section 4 provides thorough details of the implementations of the algorithms in both the CPU and the GPU. Section 5 includes a comparative study of spatial cost and rendering time. Lastly, Section 6 comments on the results obtained in our tests.

2 Related Work

Evolution of graphics hardware has given rise to new techniques that allow us to accelerate multiresolution models. This research field has been exploited for many years, and it is possible to find a wealth of papers which present very different solutions. Nevertheless, the authors have lately re-oriented their efforts towards the development of new models which consider the possibilities offered by new graphics hardware. Recent GPUs include vertex and fragment processors, which have evolved from being configurable to being programmable, allowing us to execute shader programs in parallel.

In general, multiresolution models can be classified into two large groups [3]: discrete models, which contain various representations of the same object, and continuous models, which represent a vast range of approximations.

With respect to discrete models, they offer a very efficient solution but they usually present visual artifacts when switching between pre-calculated levels of detail. A possible solution to avoid these *popping* artifacts is the use of geomorphing [4] or blending [5] in the GPU. A more thorough method is that presented in [6], which consists in sending to the GPU a mesh at minimum level of detail and applying later a refining pattern in the GPU to every face of the model. The problem, according to the authors, is that it suffers again the popping effects. Another aspect is the load suffered by the GPU when a model keeps the level of detail, as a pass must be made for each face that the coarser model has.

It is possible to find in the literature continuous algorithms aimed at rendering common meshes by exploiting GPUs. The LodStrips model was reconsidered in [7] to offer a GPU-oriented solution, by creating efficient data structures that can be integrated into the GPU. Ji et al. [8] suggest a method to select and visualize several levels of detail by using the GPU. In particular, they encode the geometry in a quadtree based on a LOD atlas texture. The main problem of this method is the costly process that the CPU must execute in every change of level of detail. Moreover, if the mesh is too complex, the representation with quadtrees can be very inefficient and even the size of the video memory can be an important restriction. Finally, the work presented by Turchyn [9] is based on Progressive Meshes. It builds a complex hierarchical data structure that derives in great memory requirements. Moreover, it changes the mesh connectivity trying to reduce memory costs.

Many of the GPU-based continuous models are aimed at view-dependent rendering of massive models. Works like [10],[11],[12] have adapted their data structures so that CPU/GPU communication can be optimized to fully exploit the complex memory hierarchy of modern graphics platforms. With a similar objective but with a further GPU exploitation, the GoLD method [13] introduces a hierarchy of geometric patches for very detailed meshes with high-resolution textures. The maintenance of boundaries is assured by means of geomorphing performed in the GPU. Finally, the work presented in [14] introduces a multi-grained hierarchical solution which avoids the appearance of cracks in the borders of nodes at different LODs by applying a border-stitching approach directly in the GPU.

In general, discrete models are easier to be implemented in GPUs but they do not avoid the popping artifacts. By contrast, continuous models offer a better granularity and avoid that problem, although their memory requirements are high and some of them even need several rendering passes for one LOD change.

3 Our Approach

The solution we are presenting offers an easy and fast level-of-detail update which, contrary to previous multiresolution models, modifies the list of vertices instead of the indices. The basic idea is to order indices and vertices so that we can apply a sliding-window approach to the level-of-detail extraction process.

Before explaining in detail the basis of our proposed solution, it is important to comment on the simplification algorithm that we will use for obtaining the sequence of approximations of the original model. This sequence will be used to obtain a progressive coarsening (and refinement) of the original model.

3.1 Mesh Simplification

It is possible to find plenty of research on appearance-preserving simplifications methods. The most important contributions have been made in the areas of geometric-based algorithms[15],[16] and viewpoint-based approaches[17],[18],[19]. Among these works, we will use an edge-collapse based method which preserves texture appearance [19]. It is important to comment that in these collapse operations we will not modify vertices coordinates, as we assumed that the vertex that disappears will collapse to an existing one. The selection of this type of edge-collapse simplifies the data structures of our model and still offers very accurate

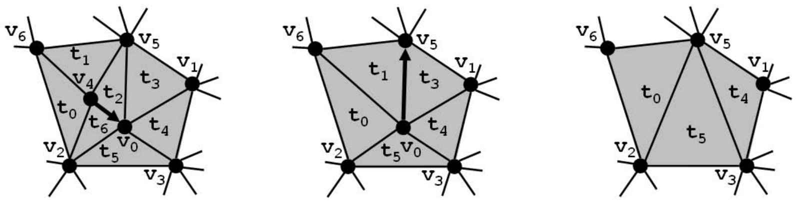

Fig. 1. Simplification of a section of a polygonal model

simplifications. An example of this simplification process can be observed in Fig. 1, where a section of a polygonal mesh is simplified with two edge-collapse operations.

3.2 Sliding-Tris Framework

This multiresolution model represents a mesh with three different sets. Let M be the original polygonal surface and V and T its sets of vertices and triangles. We will also define the set E, which refers to the evolution of each vertex and will be explained later. Considering that n is the number of vertices and m is the number of triangles, $M = \{V, T, E\}$ can be defined as:

$$V = \{v_0, v_1, ..., v_n\}, T = \{t_0, t_1, ..., t_m\}, E = \{e_0, e_1, ..., e_n\} \tag{1}$$

As we have previously commented, the main idea of Sliding-Tris is to update the contents of the vertices list instead of the indices one. As an example, collapsing vertex i to vertex j would mean that the coordinates values of vertex i will be replaced with the values of vertex j.

The multiresolution model we are presenting is also based on the adequate ordering of vertices and triangles:

– Vertices: they are ordered following the collapse order, so that vertex i will collapse when changing from lod $i-1$ to lod i.
– Triangles: they are ordered according to their elimination order, so that the last triangle will be the first one to disappear.

Following with the simplification example offered in the previous section, the correct ordering of the initial mesh should be the one presented in Fig. 2. On top we present the original contents of the triangles and vertices lists. In the bottom, we offer the ordered lists obtained following our requirements. Thus, we can see how vertex 4 is now vertex 0 and vertex 0 is now vertex 1, following the order of vertex collapses. In a similar way, triangles 6 and 2 are now the last ones, as they will be the first ones to disappear.

As we already commented, we will need to store the evolution of each vertex. The evolution reflects the different vertices an original one collapses to throughout

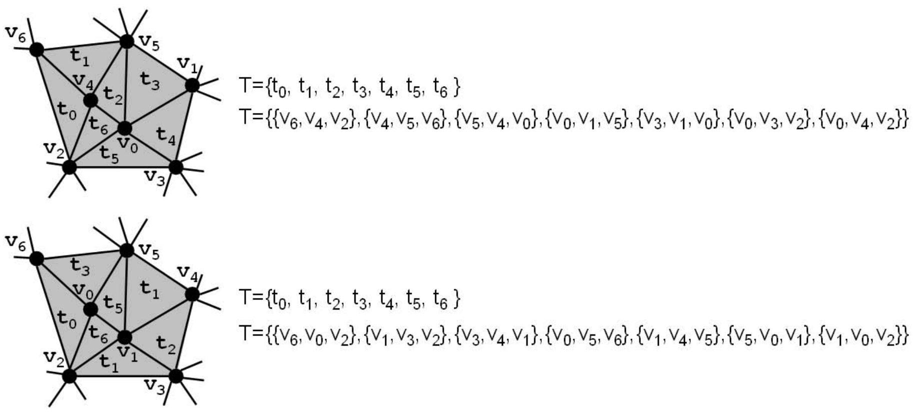
T={t0, t1, t2, t3, t4, t5, t6 }
T={{v6,v4,v2},{v4,v5,v6},{v5,v4,v0},{v0,v1,v5},{v3,v1,v0},{v0,v3,v2},{v0,v4,v2}}
T={t0, t1, t2, t3, t4, t5, t6 }
T={{v6,v0,v2},{v1,v3,v2},{v3,v4,v1},{v0,v5,v6},{v1,v4,v5},{v5,v0,v1},{v1,v0,v2}}

Fig. 2. Initial ordering (top) and re-order of triangles and vertices (bottom)

```
// LOD Extraction algorithm.
for (vert=0 to demandedLOD ) {
  i = 0;
  while (Evolution[vert][i] < demandedLOD)
    i++;
  CopyVertex(CurrentVertices[vert],OriginalVertices[i]);
}
// Visualization algorithm.
numTriangles -= 2*(demandedLOD-currentLOD); //If increasing detail add.
glDrawElements (Triangles,0,numTriangles,...);
```

Fig. 3. Pseudocode of a simple CPU implementation of the LOD algorithms

the levels of detail. Thus, each e_i element will be composed of a list of references to the vertices that vertex i collapses to. As the vertices have been ordered following the collapse order, we will be able to know in which LOD a particular vertex must change. More precisely, we can assure that the evolution of vertex i satisfies that we must use the contents of its j-th element while $e_{i,j} \leq demandedLOD < e_{i,j+1}$.

Finally, we will also have to store a copy of the original vertices, which will be used for updating the value of each vertex when traversing the different LODs.

Once we have fulfilled all these requirements, we are ready to start with the algorithm that will enable us to obtain all the levels of detail (Fig. 3). Each time we change to a different LOD we must check every vertex to see if it is necessary to update it. Nevertheless, due to the order we have chosen for the vertices, it will only be necessary to check vertices from 0 to $demandedLOD - 1$. Once we have updated the necessary vertices, the sliding-window approach is applied to render the suitable number of indices.

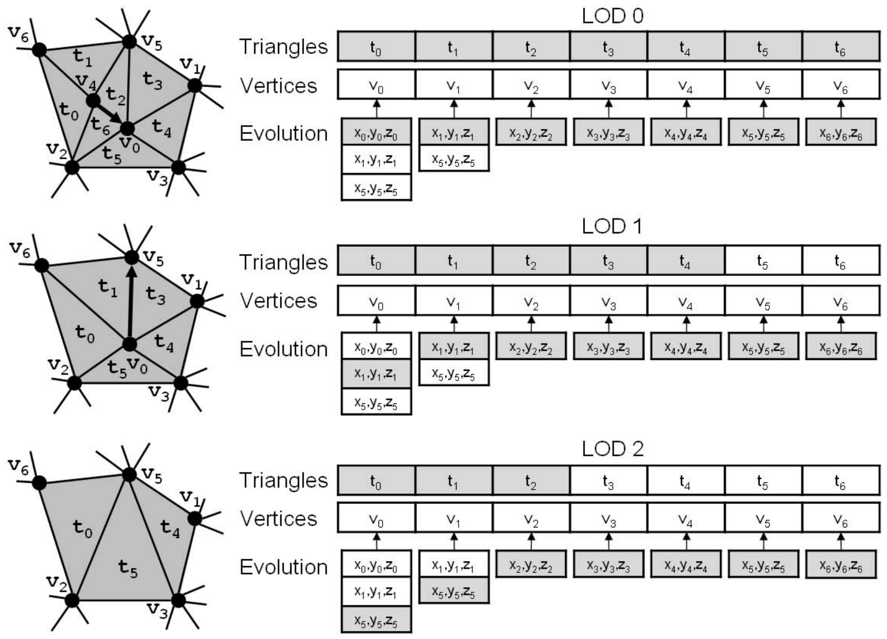
LOD 0
Triangles
Vertices
Evolution
LOD 1
LOD 2
t_0 t_1 t_2 t_3 t_4 t_5 t_6
v_0 v_1 v_2 v_3 v_4 v_5 v_6
x_0,y_0,z_0
x_1,y_1,z_1
x_5,y_5,z_5

Fig. 4. Example of the extraction process of three levels-of-detail

Fig. 4 presents the evolution of the example mesh during three edge collapses. This figure includes, for each level of detail, the array of triangles and vertices and, for each vertex, its evolution. The array of triangles is shaded following the sliding-window approach. With respect to the evolution, the shaded cell reflects the current contents of the vertex. Following the algorithm introduced in Fig. 3, let's suppose we change from LOD 0 to LOD 1. We would decrease the triangle count bt two and, in this case, we would only modify vertex 0 so that its coordinates are updated with the contents of vertex 1. For the second LOD change, vertices 0 and 1 must be updated, and according to the contents of their evolution, they must change their values for vertex 5.

4 Sliding-Tris Implementation

In this section we will introduce different possible implementations of the original algorithm which will allow us to exploit the graphics hardware. The first version will store the mesh information in the GPU, updating it in the most appropriate way. Nevertheless, this solution still involves data traffic through the BUS. As a consequence, we will also present the GPU implementations of the algorithm in both the vertex shader and the pixel shader, which reduce the traffic to just uploading the value of the new *demandedLOD*.

CPU Version. Storing the information in buffers in the GPU offers a faster rendering. Thus, this version uploads indices and vertices to the GPU. It is important to comment that we must keep a copy of the current vertices and also the original ones in the CPU, as well as the evolution of the vertices. The extraction process is similar to that presented in Fig. 3, but once we have updated the array of vertices in the CPU, we will upload it to GPU. More precisely, we will carefully delimit the vertices to transfer to minimize traffic. We can then render the mesh by indicating how many triangles must be considered.

Exploiting the Possibilities of the Vertex Shader. For adapting the algorithm to shaders in the GPU, the first problem we must address is how to store the necessary information (the evolution and the original vertices) in the GPU. On the one hand, we create a floating point texture to store the original values of the vertices. On the other, the evolution of each vertex will be stored in different sets of its attributes, mainly in unused *MultiTexCoords*. These vertex attributes can store 4 components, which in our case represent 4 elements of the evolution. We will use as many attributes as necessary. For efficiency, we store all the vertex attributes in a single interleaved array.

Once the data is stored, the only information that the CPU must send to the GPU is the new LOD value. The original extraction process has been carefully adapted to work on a vertex shader, consulting the attributes to analyze the evolution and accessing the texture once the correct value of the evolution is obtained. Thus, each vertex uses this shader to correctly update its coordinates.

An important issue with this version is that we oblige the vertex shader to update the coordinates even when the LOD is maintained, as we are not storing the resulting vertex buffer. This is an important disadvantage, but we must consider than in interactive applications the user keeps moving all the time, and under these conditions the detail must be updated very often.

Exploiting the Possibilities of the Pixel Shader. To overcome the limitation of the vertex shader implementation, we adapted the Sliding-Tris algorithm to a pixel shader. In this case, we will use a render-to-vertex approach to store the newly calculated vertices.

With respect to storing the necessary data, we will still use a texture to store the initial values of the vertices, but we will create a new one to store the evolution of the vertices.

The main algorithm must consider the render-to-vertex operation[20]. This way, before rendering the model we will define a viewport and render a quad that fills it, covering as many pixels as vertices we must update. Thus, each pixel will use the shader to compute the value of a different vertex. This pixel shader will use a routine similar to the vertex shader but, in this case, consulting the evolution of a vertex implies accessing a texture instead of the attributes. Once all pixels have been evaluated, the CPU must perform an extra operation which involves reading the output buffer into a *VertexBufferObject* via *ReadPixels*. Then, we can disable this pixel shader and render the mesh normally using this buffer object as a new source of vertex data.

Table 1. Models used in the experiments, with their storing cost (in MB.)

Model	Cow	Bunny	Dragon	Phone	Isis	Buddha
Vertices	2904	35947	54296	83044	187644	543644
Faces	5804	69451	108588	165963	375283	1085634
Original (triangles)	0.10	1.21	1.86	2.84	6.44	18.65
Progressive Meshes	0.27	3.28	5.09	7.86	17.23	51.28
LodStrips	0.17	2.21	3.32	5.08	11.69	35.51
Sliding-Tris	0.12	1.45	2.23	3.45	7.72	22.56

5 Results

In this section we will present some tests that cover the storing cost and the rendering performance of the presented versions of Sliding-Tris. The experiments were carried out using Windows XP on a PC with a processor at 2.8 Ghz, 2 GB RAM and an nVidia GeForce 7800 graphics card with 256MB RAM. The different implementations have been done in C++, OpenGL and HLSL.

5.1 Storing Cost

Table 1 shows a comparison of spatial costs among previous continuous uniform resolution models: PM [21], a triangle-based approach, and LodStrips [7], which is based on triangle strips. As it can be observed, the model presented offers the best spatial cost. On average, it fits in 1.2 times the original mesh in triangles, in contrast to PM and LodStrips which fit in 2.7 and 1.9 times respectively. This is due to the fact that the only extra information that we store is the evolution of the vertices. Furthermore, it is important to note that the size of each evolution (e_n) element is usually small. Our experiments have shown that the evolution of the vertices of most models have a maximum number of 12 elements and an average of 2, despite being meshes composed of thousands of vertices.

5.2 Rendering Time

We analyzed the frame rate obtained throughout the different levels-of-detail when rendering a bunny model without textures nor illumination. We have also included the results of a version without extraction cost, in order to show which frame-rate could be obtained at most. The results are presented in Fig. 5, where it can be observed that the frame rate of the pixel shader version is always the lowest, as this version obliges the pipeline to stop until the render-to-vertex has been completely performed, thus limiting the performance of both the CPU and the GPU. The best results are offered by the vertex shader implementation. It is important to note that in this test we extracted a new LOD for each frame. Thus, the CPU version and the pixel shader one would be able to achieve the performance of the original model when the LOD remains stable. As commented before, interactive applications tend to change scene conditions permanently, and under these circumstances the vertex shader offers the best performance.

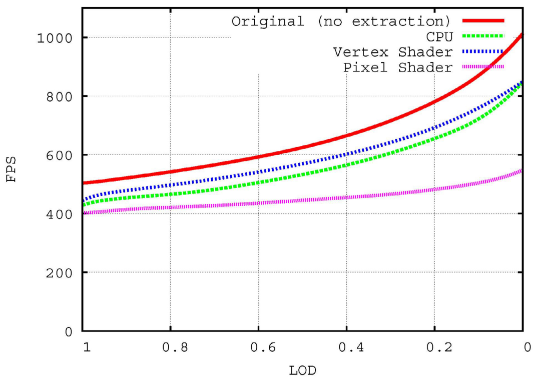

Fig. 5. Frame rate obtained when rendering the bunny model with the different implementations. The *original* values refer to the visualization cost only.

6 Conclusions

In this paper we have presented a new multiresolution model which has been completely adapted to the GPU. Sliding-Tris offers a low storing cost, easy implementation and a fast extraction process which make it suitable for any rendering engine. A further advantage is that the extraction process is always similar in cost. The shaders must consider the extraction process for each vertex, and, as a consequence, these algorithms would obtain similar results when the difference between the demanded and the current LOD is big or small, in contrast to hierarchical models.

For the CPU model, an important limitation is the data traffic involved in extracting the approximations. Updating vertices instead of indices involves working with three floats per vertex instead of one integer per index. This limitation can be worse if the meshes work with normals, textures, etc. Nevertheless, our experiments have shown that the total number of update operations is similar, and the final rendering speed is not affected by this increase in the quantity of data interchanged.

The use of shaders to perform the LOD changes avoids the traffic problem, even though the render-to-vertex approach is still slow and demands CPU intervention. As a consequence, even when the three approaches are quite similar in rendering time, the vertex shader offers a wiser solution as it can be running while the CPU and GPU are performing a different operation.

Acknowledgments. This work has been supported by grant P1 1B2007-56 (Bancaixa), the Spanish Ministry of Science and Technology (Contiene Project: TIN2007-68066-C04-02) and FEDER funds.

References

1. Clark, J.: Hierarchical geometric models for visible surface algorithms. CACM 10(19), 547–554 (1976)
2. Garland, M., Heckbert, P.: Simplification using quadric error metrics. Computer and Graphics 31, 209–216 (1997)
3. Ribelles, J., Chover, M., Lopez, A., Huerta, J.: A first step to evaluate and compare multiresolution models. In: EUROGRAPHICS, pp. 230–232 (1999)
4. Sander, P.V., Mitchell, J.L.: Progressive buffers: View-dependent geometry and texture for lod rendering. In: Symp. on Geom. Process, pp. 129–138 (2005)
5. Southern, R., Gain, J.: Creation and control of real-time continuous level of detail on programmable graphics hardware. Comp. Graph. For. 22(1), 35–48 (2003)
6. Boubekeur, T., Schlick, C.: Generic mesh refinement on gpu. In: Graphics Hardware, pp. 99–104 (2005)
7. Ramos, F., Chover, M., Ripolles, O., Granell, C.: Continuous level of detail on graphics hardware. In: Kuba, A., Nyúl, L.G., Palágyi, K. (eds.) DGCI 2006. LNCS, vol. 4245, pp. 460–469. Springer, Heidelberg (2006)
8. Ji, J., Wu, E., Li, S., Liu, X.: Dynamic lod on gpu. In: CGI (2005)
9. Turchyn, P.: Memory efficient sliding window progressive meshes. In: WSCG (2007)
10. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In: SIGGRAPH, pp. 796–803 (2004)
11. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: Batched multi triangulation. In: IEEE Visualization, pp. 207–214 (2005)
12. Yoon, S., Salomon, B., Gayle, R.: Quick-vdr: Interactive view-dependent rendering of massive models. IEEE Transactions on Visualization and Computer Graphics 11(4), 369–382 (2005)
13. Borgeat, L., Godin, G., Blais, F., Massicotte, P., Lahanier, C.: Gold: interactive display of huge colored and textured models. Trans. Graph. 24(3), 869–877 (2005)
14. Niski, K., Purnomo, B., Cohen, J.: Multi-grained level of detail using a hierarchical seamless texture atlas. In: Proceedings of I3D 2007, pp. 153–160 (2007)
15. Cohen, J., Olano, M., Manocha, D.: Appearance-preserving simplification. In: SIGGRAPH 1998, pp. 115–122. ACM Press, New York (1998)
16. Gonzalez, C., Gumbau, J., Chover, M., Castello, P.: Mesh simplification for interactive applications. In: WSCG (2008)
17. Lindstrom, P., Turk, G.: Image-driven simplification. ACM Trans. Graph. 19(3), 204–241 (2000)
18. Luebke, D., Hallen, B.: Perceptually-driven simplification for interactive rendering. In: 12th Eurographics Workshop on Rendering, pp. 223–234 (2001)
19. Castello, P., Chover, M., Sbert, M., Feixas, M.: Applications of information theory to computer graphics (part 7). In: Eurographics Tutorial Notes, Eurographics, vol. 2, pp. 891–902 (2007)
20. Biermann, R., Cornish, D., Craighead, M., Licea-Kane, B., Paul, B.: pixel buffer objects (2004), `http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_pixel_buffer_object.txt`
21. Hoppe, H.: Progressive meshes. In: SIGGRAPH, pp. 99–108 (1996)