View-Dependent Multiresolution Modeling on the GPU

Oscar Ripolles, Jesus Gumbau, Miguel Chover, Francisco Ramos, Anna Puig-Centelles
LSI Department
Universitat Jaume I, Castellón, SPAIN
{oripolle, jgumbau, chover, jromero, apuig}@uji.es

ABSTRACT

For more than a decade, researchers working on level-of-detail techniques have oriented their efforts toward developing better frameworks and adapting their solutions to new hardware. Nevertheless, we believe there is still a gap for efficient yet simple multiresolution models that fully exploit the potential of current GPUs. In this paper we present a level-of-detail framework based on moving the extraction process from updating indices to updating vertices. This feature enables us to perform culling and geomorphing on a vertex basis. Furthermore, it simplifies the update of indices to eliminate degenerate information. The model is capable of offering both uniform and variable resolution and to achieve the latter, a silhouette-based criterion has been included. Finally, we would like to highlight that the model is completely integrated in the GPU and no CPU/GPU communication is necessary once all the information is correctly loaded in hardware memory.

Keywords: Multiresolution modeling, View-dependent, Silhouette-preserving, Shader Model 4.0

1 INTRODUCTION

Multiresolution modeling has been successfully applied to solve problems in many areas [17] and there is an important body of literature on the subject [15]. The problem with traditional level-of-detail (LOD) techniques is the fact that they usually include complex data structures and algorithms which are difficult to translate to GPU. This complexity is the reason why LOD methods are traditionally performed on the CPU.

The development of Shader Model 4.0 was a breakthrough in computer graphics as it offers a new range of functionalities [1]. The main contribution is the *Geometry Shader*, which establishes a new stage inside the graphics pipeline enabling the dynamic creation and elimination of geometry in the GPU. Furthermore, it also offers the possibility of modifying the flow of information by means of the *Stream Output*.

The Shader Model 4.0 offers a new opportunity for the development of extremely fast level-of-detail methods. What we propose in this paper is a new model that combines the computational power of GPUs with the wealth of work already done in multiresolution. The main objective is to develop a variable resolution model that preserves appearance and avoids popping artifacts while offering high performance. More precisely, the model that we are presenting has the following features:

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.

- **Fully GPU-based implementation.** The solution is completely integrated in the GPU, exploiting the recent Geometry Shader for storing the calculated levels-of-detail and eliminating degenerate triangles. In addition, our method is capable of offering both uniform and variable resolutions, which can be calculated in GPU with no CPU intervention. The Vertex Shader will select the optimal level-of-detail to be extracted for the particular viewing conditions.
- **Updating vertices and indices in the GPU.** The solution introduced in [21] presented a promising framework where the level-of-detail transitions modify vertices instead of indices. Traditional models have always updated the information related to the indices. In this work we will update vertex coordinates to reflect LOD changes in a Vertex Shader while we will modify the indices list to delete degenerate triangles in a Geometry Shader. This approach integrates well with other pixel-based methods like sub-surface scattering or parallax occlusion mapping.
- **Optimization of visual quality.** It is also important to ensure that the original appearance of the model is kept. With this aim, we will offer variable resolution with a silhouette-based criterion. Moreover, we will perform geomorphing between the collapsed vertices in order to avoid disturbing effects like discontinuities or popping artifacts. Figure 1 presents several visualizations of a model of a man at different levels of detail using a silhouette-preserving extraction algorithm.
- **Use of triangles as the rendering primitive.** The use of triangles as rendering primitive limits performance, compared to triangle strips. Nevertheless, cache-aware optimizations can considerably

 ISBN 978-80-86943-93-0

Figure 1: Approximations of a man model (136,410 triangles). From left to right: original model and simplifications to 50%, 25% and 10% respectively.

improve the final performance [5, 11]. In our case we have applied the method presented in [18] which is based on one of the latest methods [23].

The structure of this paper is as follows. Section 2 contains a review of the work previously carried out on GPU-friendly multiresolution modeling. Section 3 presents the basic framework of our method. Section 4 provides thorough details of the implementations of the algorithms in the GPU. Section 5 includes a study of the performance of our model. Lastly, Section 6 comments on the results obtained in our tests.

2 RELATED WORK

Much work has been published on multiresolution modeling. In this section we will focus on the lines of work that are currently active in the level-of-detail field which are oriented toward the exploitation of GPUs.

Firstly, we believe it is important to mention the works designed to exploit the complex memory hierarchy of modern graphics platforms. In this sense, they develop "GPU-friendly" static vertex and index buffers and try to optimize their use by minimizing data transfers between CPU and GPU [19, 20, 24]. This idea has also been applied to rendering massive models in real time [6, 7, 22, 26].

Following on with massive models, many researchers have recently proposed methods for moving the granularity of the representation from triangles to triangle patches [2, 6, 7, 26] in order to offer view-dependent capabilities for rendering out-of-core models. Preserving boundaries is a key feature of these algorithms and it is possible to find algorithms that propose GPU-based solutions by means of geomorphing [2] or border-stitching techniques [16].

Furthermore, it is also possible to find algorithms that propose a progressive creation of geometry in the GPU [3, 9, 13]. These models offer interesting results although they are not aimed at rendering meshes in real-time applications.

Recently, the work proposed in [21] presented a GPU multiresolution model which updated vertices instead of indices and used a fixed order of triangles that permitted the use of a sliding-window scheme. Nevertheless, this fixed order simplified the algorithm while limited it to perform further extensions.

Finally, it is important to comment on those methods which use the silhouette as their criterion for extracting the desired approximation. Silhouettes are particularly important to offer realistic visualization. Many authors have presented silhouette-preserving variable multiresolution approaches [10, 25]. More recently, the work presented in [12] introduces a GPU-based adaptive model for non-photorealistic rendering. They propose a hierarchical multiresolution model and use the GPU to refine the areas around the silhouettes. Later, Dyken et al. [9] introduced a framework for calculating silhouettes on the GPU and tessellating afterwards those areas that need further detail. The main drawback of this approach is the fact that the process for calculating the silhouettes is complicated and requires several rendering passes.

3 OUR FRAMEWORK

The method we are presenting will perform the LOD update in a Vertex Shader and the degenerate filter in a Geometry Shader. As a consequence, we will be able to apply the whole extraction process in only one pass. The outputted geometry will be ready for passing through the Pixel Shader to generate the final image. It is worth mentioning that it would be possible to use the Stream Output method to store the calculated approximation to use it in subsequent renderings. Figure 2 shows a diagram of the different processes that will take place in each rendering stage.

3.1 Simplifying the original mesh

For our multiresolution model we decided to use an edge-collapse simplification algorithm to obtain the hierarchy of collapses. We could use any simplification

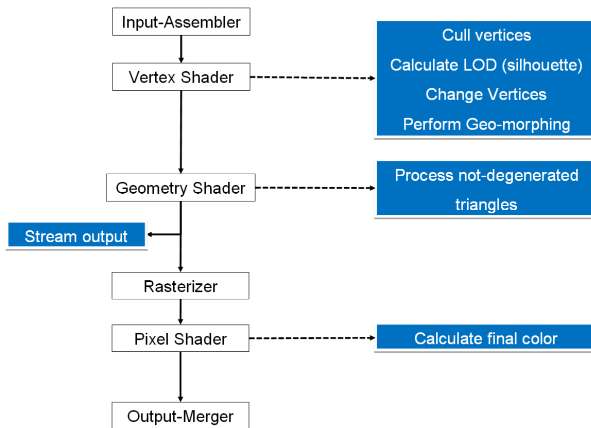

Figure 2: Rendering pipeline for our approach.

algorithm [4, 8, 14]. Our solution is not restricted to use half-edge collapses where we do not add new vertices. The only restriction is to pre-calculate the vertices so that we can store them properly before starting to use our multiresolution model. Nevertheless, in the examples that we present in this paper we will assume that the half-edge collapse is the selected simplification operation as it offers a less complex implementation.

3.2 Ordering of vertices

As we commented in the introduction, the work we are proposing is based on the ideas presented in [21]. This work introduced some basic ideas for managing collapse information, which are:

- Ordering the vertices in collapse order, so that vertex v will be collapsed when changing from LOD $v-1$ to LOD v.
- Calculating the evolution of each vertex, which contains a list of all the vertices that the vertex will collapse to.

In order to clarify this construction process, Figure 3 presents a section of the collapse hierarchy of one of our test models after correct ordering of the vertices. The evolution of each vertex stores the branch of this tree that links it with the root node. Thus, for example, the evolution of vertex 18 will be a list of indices to vertices composed of values $(38,49)$. These values will indicate at which LOD we must perform a change and, in addition, which change should be performed. Thus, following on with the example we can say that we must perform the collapse $18 \rightarrow 38$ when changing to LOD 38, and that we must apply collapse $38 \rightarrow 49$ when swapping to LOD 49.

It is important to comment that the presented approach offers a truly selective refinement, where we can apply any collapse without applying further collapses or other requisites. As the simplifications are applied in a vertex-basis, all the triangles sharing that vertex will be modified in the same way. Figure 4 shows an image of a Beethoven model simplified with this method, where half of the model is simplified to 80%. It can be seen how no crack or other disturbing effect is produced, even though there is a severe change of resolution.

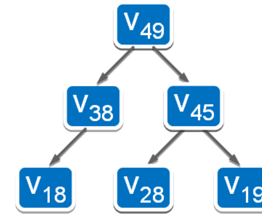

Figure 3: Example of the collapse hierarchy of a sample model.

3.3 Optimizing the rendering primitive

The solution presented in [21] proposed a sliding-window approach as it entails a fixed order of triangles which limits further extensions. This limitation is overcome in our model with the use of primitives optimized for the vertices cache [18], which orders the indices in an optimized way which renders much faster than the triangles ordered in eliminations fashion.

The pre-ordered list of triangles assured that no degenerate triangle is rendered. In our framework, we will control the appearance of degenerate information directly in a Geometry Shader.

4 IMPLEMENTATION DETAILS

In this section we will address thoroughly the shaders that we have developed. In Figure 5 we present a detailed description of the implemented shaders using a GLSL-like pseudocode. The Vertex Shader will be used to update the level-of-detail following the silhouette criterion. In the Geometry Shader we will mainly perform the degenerate triangles elimination. We will later address the possibilities offered by the Stream Output for our multiresolution model.

4.1 Storage of information

Before describing the implemented process, it is important to clarify the way the information is stored. The model we are presenting has very low memory requirements. The only extra information that we will need to store is the information about the evolution of each vertex.

The information of the original mesh (vertex coordinates, texture information, normals and so on) will be stored in floating point textures. This information will be accessed if necessary from the Vertex Shader.

 ISBN 978-80-86943-93-0

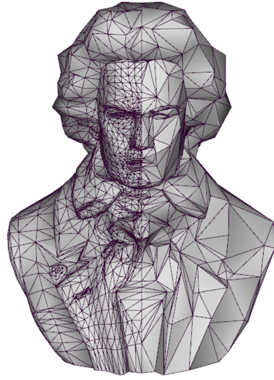

Figure 4: Beethoven model with its right side simplified to 80%.

The Shader Model 4.0 permits defining and using non-squared textures without the restriction of having a size power of two, offering a more cost-effective storage of the information.

The evolution of the vertices will be stored in different sets of attributes. The Vertex Shader will be in charge of obtaining the real vertex information so that the final geometry is correctly rendered.

4.2 Vertex Shader

The Vertex Shader will be responsible of calculating the appropriate LOD, updating vertices information and performing geomorphing.

```
// Vertex Shader
uniform LOD;
varying newID;
float3 newCoords,nextCoords;

calculateAngle(view,normal);
calculated_lod=interpolate(LOD,angle);
newCoords=
    getNewCoordinates(gl_VertexID,lod);
nextCoords=
    getNextCoordinates(gl_VertexID,lod);
newID=getNewID(gl_VertexID,lod);
nextID=getNextID(gl_VertexID,lod);
gl_Position=
    geomorph(newCoords,nextCoords,newID);

// Geometry Shader
varying newID;

if (isTriangle(newID[0],newID[1],newID[2]))
  outputTriangle();
```

Figure 5: Pseudocode of the implemented shaders.

The first instructions of the Vertex Shader will calculate the angle between the view vector and the normal of the vertex. The Vertex Shader has full access to the ModelView matrix. As a consequence, we can easily calculate the dot product between the vertex normal and the view direction.

We want to develop a model that will not need any information from the CPU once the model is correctly loaded into GPU memory. With that aim, we want to calculate the appropriate LOD according to the scene conditions inside the GPU. Knowing the angle between the vector that points towards the camera and the normal of the vertex will allow us to easily perform a silhouette-based extraction process. In those cases where the vectors are nearly perpendicular, we will need to render highly-detailed geometry to obtain the visual perception of the silhouette. In those cases where the vectors are nearly parallel, we will simplify the vertices as they do not contribute to the silhouette. In the rest of cases, we will perform a linear interpolation so that the geometry refines progressively towards the areas of the silhouette.

After these different steps, we are able to extract the correct geometry for the scene conditions. We will consult the evolution information to know which vertex information must be used, recovering all the information from the previously-defined textures.

It is important to note that we will recover the information of the vertex that we currently need and the following one. With the two extracted vertices we can make some simple calculations to assure a progressive transition among LODs. The way that the evolution is stored will assure that a vertex will collapse to its j-th element of the evolution once we reach LOD j. Our proposal is to geomorph between vertices stored in the positions $j-1$ and j while the LOD value is contained between $j-1$ and j. Thus, once we reach LOD j the vertex will be completely changed to vertex j, ensuring that the collapse information is correctly applied. With this approach the continuity of the mesh is ensured, as all the vertices will be collapsed in the same way. Following with the example given in Figure 3, if we are at LOD 48 the vertices 28, 19 and 45 will contain the same interpolated value ensuring mesh continuity.

Finally, the Vertex Shader, in addition to outputting the vertex information, will also output the ID of the vertex we have collapsed to.

The LOD traversal algorithm is efficient and is capable of extracting any level-of-detail with the same cost. Thus, making big LOD changes is not penalized. Moreover, the Pixel Shader is not necessary and remains available for any further extension that we may wish to apply.

4.3 Geometry Shader

The development of the Geometry Shader has made it possible to work directly with triangles in a new stage. This feature is very powerful but the Geometry Shader

is not a stage that must be activated. Consequently, the use of Geometry Shaders involves slowing down the whole rendering process. Nevertheless, it is worth activating this rendering stage when we are able to discard a considerable amount of geometry or when we need to create geometry on-the-fly. In our case, we can expect that the more coarse the approximation that we want to render, the greater the number of degenerate triangles that will be obtained. Thus, we decided to use the Geometry Shader to filter the degenerate triangles in real time. As a consequence, we will perform a simple test using the ID of the vertices output from the Vertex Shader to discard those triangles which have repeated vertices.

4.4 Exploiting Stream Output

An important drawback of the scheme we have presented is that it obliges us to make the LOD calculations for every frame, even when the level-of-detail is maintained. To overcome this limitation we can use the *Stream Output* possibilities. With this feature enabled, we are able to store the new vertices and indices, and use the modified buffers in subsequent renderings until the application updates the viewing conditions.

Storing the calculated information forces us to perform two rendering passes to obtain the correct geometry: one pass to output vertex information and another one to store the index information. Obviously, we need two new buffers to store the vertices and indices generated.

The first pass will be used to store the vertices calculated. In this case we will use no index buffer to ensure that the vertices are processed and output in the correct order.

The second pass will be the one that creates the correct index buffer. The Geometry Shader performs this calculation. We will slightly modify the shader to output a special *varying* containing, for each triangle processed, the three indices. In addition, when using the Stream Output it is possible to query the number of primitives generated from the CPU. This information, as well as the recently filled index buffer, will be used to render the model in successive frames.

In the results section we will show that the implementation of the *Stream Output* using OpenGL does not greatly affect the final performance. Nevertheless, depending on whether the application is updating the LOD in every frame or maintaining it, we can decide to switch the *Stream Output* on and off.

5 RESULTS

In this section we will present some tests that analyze the rendering performance of the model presented. The experiments were carried out using Windows Vista on a PC with a 2.8 GHz processor, 2 GB RAM and an nVidia GeForce 8800 graphics card with 256MB RAM. The different implementations have been done in C++, OpenGL and GLSL. Finally, it is important to note that we have used the GL_TIME_ELAPSED_EXT extension, which provides a query mechanism to determine the amount of time used for completing a set of GL tasks without stalling the rendering pipeline.

Table 1 presents the results obtained for a scene with a lit model of a man. This Table provides the rendering and extraction times obtained throughout three different levels of detail for given viewing conditions. The *Discrete LOD* row offers the times that would be obtained with three precalculated approximations. In this case, we have assumed that there is no extraction time. The following rows present the costs of our model, without and with the *Stream Output* extension. It can be seen how, on average, our model increases the rendering time by approximately 30%. Nevertheless, this time is necessary to perform all the tasks that are part of our view-dependent rendering pipeline. Finally, including the *Stream Output* capabilities entails slightly higher times, as we need to perform two rendering passes. Nevertheless, this extra time would allow us to reduce the extraction cost to zero in subsequent passes.

6 CONCLUSIONS

This paper has presented a new multiresolution model that combines the power of current GPUs with traditional techniques. Updating vertices instead of indices allows us to perform geomorphing among the different levels of detail to offer smooth transitions. The framework also allows for variable resolution, which can be oriented toward applying silhouette-based visualizations that better preserve the appearance of the model. This method is suitable for combining with other techniques, such as normal mapping, hardware skinning, and other pixel-based approaches.

From the results obtained we can conclude that the extraction process is expensive, as it entails increasing the final rendering time by 30%. Nonetheless, the level-of-detail extraction would be much more costly if it was applied in a CPU-based way. Furthermore, the extra cost that our model introduces is compensated by the number of calculations performed and the final visual quality.

ACKNOWLEDGEMENTS

This work was supported by the Spanish Ministry of Science and Technology with grant TSI-2004-02940 and project TIN2007-68066-C04-02. Also by Bancaja with project P1 1B2007-56.

REFERENCES

[1] D. Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

Approximation	100% (136,410 tris.)	50% (68205 tris.)	25% (34,102 tris.)
Discrete LOD	1.984	1.491	0.872
View-Dependent LOD	2.615	2.012	1.192
View-Dependent LOD + Stream Output	2.641	2.092	1.205

Table 1: Comparison of total time (extraction + rendering) for three approximations of the man model (ms.).

[2] L. Borgeat, G. Godin, F. Blais, P. Massicotte, and C. Lahanier. Gold: interactive display of huge colored and textured models. *Trans. Graph.*, 24(3):869–877, 2005.

[3] T. Boubekeur and C. Schlick. A flexible kernel for adaptive mesh refinement on GPU. *Computer Graphics Forum*, 27(1):102–114, 2008.

[4] P. Castello, M. Chover, M. Sbert, and M. Feixas. Applications of information theory to computer graphics (part 7). In *Eurographics Tutorial Notes*, volume 2, pages 891–902. Eurographics, 2007.

[5] J. Chhugani and S. Kumar. Geometry engine optimization: cache friendly compressed representation of geometry. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 9–16, 2007.

[6] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. In *SIGGRAPH*, pages 796–803, 2004.

[7] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Batched multi triangulation. In *IEEE Visualization*, pages 207–214, 2005.

[8] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In *SIGGRAPH '98*, pages 115–122, New York, USA, 1998. ACM Press.

[9] C. Dyken, M. Reimers, and J. Seland. Real-time GPU silhouette refinement using adaptively blended bézier patches. *Computer Graphics Forum*, 27(1):1–12, March 2008.

[10] H. Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, 1997.

[11] G. Lin and T. Yu. An improved vertex caching scheme for 3d mesh rendering. *Transactions on Visualization and Computer Graphics*, 12(4):640–648, 2006.

[12] Y. Livny, M. Press, and J. El-Sana. Interactive GPU-based adaptive cartoon-style rendering. *Vis. Comput.*, 24(4):239–247, 2008.

[13] H. Lorenz and J. Döllner. Dynamic mesh refinement on GPU using geometry shaders. In *WSCG*, February 2008.

[14] D. Luebke and B. Hallen. Perceptually-driven simplification for interactive rendering. In *12th Eurographics Workshop on Rendering*, pages 223–234, 2001.

[15] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, Inc., 2003.

[16] K. Niski, B. Purnomo, and J. Cohen. Multi-grained level of detail using a hierarchical seamless texture atlas. In *Proceedings of I3D'07:*, pages 153–160, 2007.

[17] E. Puppo and R. Scopigno. Simplification, lod and multiresolution - principles and applications. In *EUROGRAPHICS 1997*, volume 16, 1997.

[18] B. Purnomo. Amd tootle ver 2.0. `http://ati.amd.com/developer/tootle.html`, 2008.

[19] F. Ramos, M. Chover, O. Ripolles, and C. Granell. Continuous level of detail on graphics hardware. In *DGCI*, pages 460–469, 2006.

[20] O. Ripolles and M. Chover. Optimizing the management of continuous level of detail models on GPU. *Computers & Graphics*, 32(3):307–319, 2008.

[21] O. Ripolles, F. Ramos, and M. Chover. Sliding-tris: A sliding window level-of-detail scheme. In *Computer Graphics and Geometric Modeling (CGGM) 2008 Workshop*, 2008.

[22] P. V. Sander and J. L. Mitchell. Progressive buffers: View-dependent geometry and texture for lod rendering. In *Symp. on Geom. Process.*, pages 129–138, 2005.

[23] P. V. Sander, D. Nehab, and J. Barczak. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), August 2007.

[24] P. Turchyn. Memory efficient sliding window progressive meshes. In *WSCG*, 2007.

[25] J. Xia, J. El-Sana, and Varshney A. Adaptative real-time level-of-detail-based rendering for polygonal models. *Trans. on Visualization and Computer Graphics*, 3(2):171–183, 1997.

[26] S. Yoon, B. Salomon, and R. Gayle. Quick-vdr: Interactive view-dependent rendering of massive models. *IEEE TVCG*, 11(4):369–382, 2005.