INFORMATICA, 2013, Vol. 24, No. 4, 603–618

© 2013 *Vilnius University*

A Multiresolution Approach to Render 3D Models

Francisco RAMOS[1*], Miguel CHOVER[1], Oscar RIPOLLES[2]
[1]*Universitat Jaume I, Dept. Lenguajes y Sistemas Informaticos, 12071 Castellon, Spain*
[2]*Neuroelectrics, 08022 Barcelona, Spain*
e-mail: francisco.ramos@uji.es, chover@uji.es, oscar.ripolles@neuroelectrics.com

Received: September 2011; accepted: May 2013

Abstract. Image synthesis techniques are present in a wide range of applications as they leverage the amount of information required for creating realistic visualizations. For fast hardware rendering they usually employ a triangle-based representation describing the geometry of the scene. In this paper, we introduce a new and simple framework for performing on-the-fly refinement and simplification of meshes completely on the GPU. As we aim at making easy the integration of level-of-detail management into the creation workflow of artists, the presented method is easy to be implemented. We only need a coarse mesh, its displacement map and a geometry shader. At rendering time, we employ a geometry shader to parallelize the tessellation and displacement steps. The tessellation step performs uniform refinement or simplification operations by applying a fixed subdivision criterion. Our method also exploits coherence by taking advantage of the last computed mesh. We provide a method which offers a flexible integration with standard 3D tools, easy to be implemented, coherence exploitation and wholly processed by the GPU.

Key words: GPU, mesh refinement, shader, visualization

1. Introduction

Nowadays, realism of 3D scenes is usually proportional to their degree of geometrical complexity. Despite latest technological advances, the existing bandwidth bottleneck between applications and graphics hardware continues limiting the size of the geometric objects that can be transmitted to obtain interactive frame rates. This limitation directly affects the realism of the rendered scene.

For interactive applications, one possible solution to reduce the cost of representing a 3D scene consists in employing mesh refinement techniques. These techniques represent a surface as a coarse polygonal mesh plus a height map or displacement function which describes the difference between the original and the coarse mesh. Later, in the rendering stage, the coarse mesh suffers two consecutive operations: *tessellation* and *displacement*.

In the tessellation step, a more detailed mesh is generated while maintaining mesh topology. Then, in the displacement step, resulting vertices are traslated to their final position by sampling the displacement function. This framework offers a differential and key improvement, as the data required to describe the coarse mesh plus the displacement

[*]Corresponding author.

function is considerably smaller than the refined mesh. Subdivision surfaces, spline-based surface representation, hierarchical height and other computer graphics techniques can be expressed by means of this framework.

This kind of methods have usually been implemented on the GPU by means of vertex or pixel programs. On the one hand, the vertex shader stage offers us a straightforward implementation of the displacement step. However, in this stage, we are unable to create new geometry, which makes the tessellation step almost unapproachable. On the other hand, the fragment shader stage can also solve the displacement step by adding surface details when color texturing takes place. In general, these approaches convert a coarse mesh into a rectangular image. The tessellation step consists in an image upscaling operation and subsequently, the displacement step is performed in the fragment shader stage. Nevertheless, these methods present some disadvantages: difficulty in handling the level of detail, non-existing coherence among consecutive frames and performance penalty due to silhouettes and visibility computations.

The Shader Model 4.0 has introduced a new stage between the vertex shader stage and the rasterizer, called the geometry shader (Blythe, 2006). The geometry shader is able to generate new vertices and primitives, being the tessellation step ideally suited to be implemented within this kind of shaders. Moreover, it is also possible to implement the displacement step by modulating the tessellated mesh with a displacement function. Finally, the result can be delivered to the rasterizer unit.

In this paper, we propose an approach for refinement and simplification of triangle meshes on the GPU. It is based on the following key features:

- A flexible integration into artist's creation workflow. We employ displacement maps obtained from standard tools such as Pixologic ZBrush, `http://www.pixologic.com` or 3D Studio MAX, `http://www.autodesk.com/3dsmax` (see Fig. 1). In this way, special data formats or data structures are not required, making our approach more accessible to the final users.
- Fully-GPU refinement and simplification. We employ the geometry shader stage to move dynamic mesh refinement completely to GPU, considerably relieving the CPU. We perform both tessellation and displacement steps in this stage.
- Coherence exploitation. We take advantage of the information obtained in earlier refinement or simplification operations. Thus, we avoid the need of starting from the coarse mesh and perform the whole refinement in each frame. In this way, we are able to obtain the same rendering performances as the ones obtained with static meshes (i.e. refined during a preprocessing step and stored on the GPU).
- Avoidance of cracks and holes. A set of fixed triangle-based operations is used to avoid cracks in our refinement algorithm.

Our GPU-based refinement method does not need to transmit geometry through the graphics BUS. The coarse mesh is stored once for all on the GPU and the target mesh is completely generated in the graphics hardware. Hence, we are able to increase detail independently from CPU, which makes this work particularly interesting for applications requiring a huge refinement depth, such as CAD or scientific visualization. Moreover,

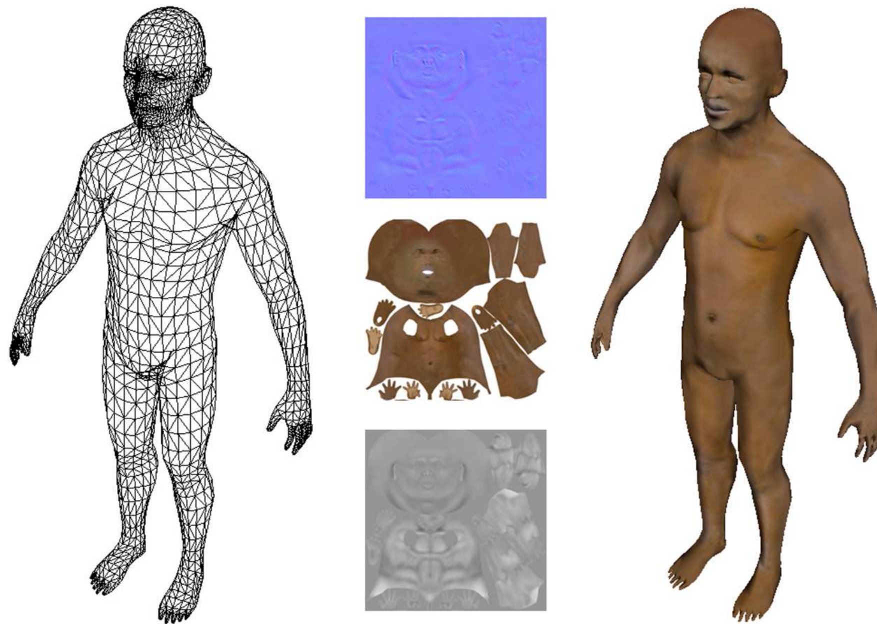

Fig. 1. Mesh refinement. On the left we offer the coarse mesh (6,364 triangles). In the middle we show its normal, texture and displacement maps. On the right we display the tessellated, displaced and lighted mesh on the GPU (1,086,122 triangles).

we make use of standard tools to generate the displacement maps in such a way that the preprocessing stage does not require costly processes or the construction of special data structures. In this sense, we also facilitate the utilization of our approach in final applications.

This paper is organized as follows. The remaining of Section 1 presents the main features of Shader Model 4.0. Then, Section 2 covers the latest work on tessellating meshes. Afterwards, Section 3 thoroughly describes our proposed framework. Section 4 presents the results obtained with our algorithms. Lastly, Section 5 concludes the performed work and proposes lines of work for extending this solution.

2. Technical Background

2.1. *Shader Model 4.0*

The Shader Model 4.0, whose pipeline is shown in Fig. 2, uses the same processor core to implement vertex, geometry and fragment processing. Separate processors with different capabilities were used for different stages of the earlier GPUs. The previous Shader Model with separate processors for vertex and pixel units was prone to under-performance. This new Shader Model dynamically allocates the available processing resources to vertex, geometry or pixel units as demanded by the load. This greatly improves the resource utilization.

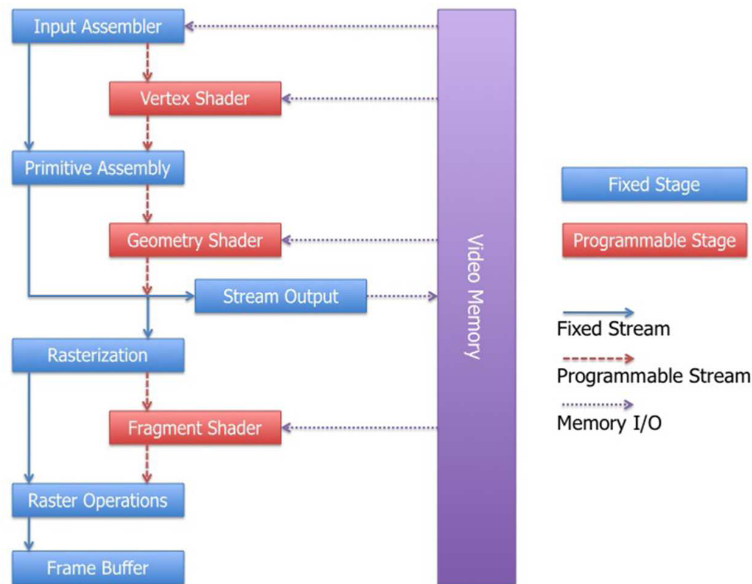

Fig. 2. The shader model 4.0 graphics pipeline.

An important advantage of this unified architecture is uniform access to the texture memory for all shaders (the vertex textures supported by Shader Model 3.0 were slow Asirvatham and Hoppe, 2005). This allows the use of textures to store regular geometry and also their access both in vertex and geometry shaders. This feature, together with geometry generation on the geometry shader, makes it possible to render huge geometry completely created on the shaders. Some of the applications which can take direct advantage of the above scenario are rendering of Geometry Images (Gu *et al.*, 2002) or rendering huge meshes or terrains with dynamic LOD. Therefore, unified shaders allow us to store geometry in form of textures and enables an efficient access to them from the shaders.

These features enable, for the whole geometry, to be stored and rendered from GPU memory. This fact can increase the rendering speed especially if the geometry is quite regular. Terrains are good examples of such geometry when represented using regular heightmaps. The 2D array of heights can be stored in textures on the GPU and accessed by the shaders quickly for rendering (Szirmay-Kalos and Umenhoffer, 2008). These concepts parallel our work, although we aim at applying them to the refinement of arbitrary triangular meshes.

2.2. *Mutiresolution Models*

One of the main objectives of multiresolution models consists of minimizing the number of polygons that are sent to the graphics pipeline without affecting the visual quality of the resulting image.

The simplest method to create a multiresolution model is to generate a fixed set of approximations. In a given instant, the graphics application could select the approximation

Fig. 3. From left to right, original object and three levels of detail of the ogre model.

to visualize. In this case, a series of discrete levels of detail would be used, as seen in Fig. 3. This multiresolution model would consist of a set of levels of detail and some control parameters to change between them.

Instead of creating individual levels of detail, continuous multiresolution models present a series of continuous approximations of an original object. The simplification method employed offers a continuous flow of simplification operations to progressively refine the original mesh.

The main advantage of these models is their better granularity, that is, the level of detail is specified exactly, and the number of visualized polygons is adapted to the requirements of the application. This granularity is usually of a few triangles, as the difference between contiguous levels of detail is usually of a vertex, an edge or a triangle. Moreover, the spatial cost is lower since the information is not duplicated. Obviously, the management of the level of detail in these models is an essential point, that is, the amount of time required to visualize a level of detail should not never exceed the time required to visualize the object at its maximum resolution.

One of the first models to offer a neat solution to a continuous representation of polygonal meshes was Progressive Meshes (Hoppe, 1996). It simplifies a mesh $M = M^n$ in consecutive approximations M^i by applying a sequence of n edge collapses:

$$M = M^n \xrightarrow{Collapse_{n-1}} M^{n-1} \xrightarrow{Collapse_{n-2}} \cdots \xrightarrow{Collapse_0} M^0. \tag{1}$$

The reverse operation can also be performed, recovering more detailed meshes from the simplest mesh M^0 by using a sequence of *vertex splits*:

$$M^0 \xrightarrow{Split_0} M^1 \xrightarrow{Split_1} \cdots \xrightarrow{Split_{n-1}} M^n = M. \tag{2}$$

3. Related Work

Height fields are natural representations in a variety of contexts, including water and terrain modeling. In these cases, height maps are provided by simulation or measurement processes. They are gray scale images and can thus also be generated by 2D

drawing tools (Pixologic zbrush, `http://www.pixologic.com`; Discreet 3D Studio MAX, `http://www.autodesk.com/3dsmax`) (see Fig. 1). They can also be the result of surface simplification when the difference between the detailed and coarse surface is computed (Cignoni *et al.*, 1998; Blasco, 2002; Ati normalmapper tool, `http://ati.amd.com/developer/tools.htmlAti03`). In this way, height and normal maps construction are closely related to tessellation (Gumhold and Hüttner, 1999; Doggett and Hirche, 2000; Doggett *et al.*, 2001; Moule and Mccool, 2002; Espino *et al.*, 2005) and subdivision algorithms (Catmull, 1974; Boubekeur and Schlick, 2005). Displacement maps can also be the result of rendering during impostor generation, when complex objects are rasterized to textures in order to be displayed instead of the original models (Jeschke *et al.*, 2005). By copying not only the color channels but also the depth buffer, the texture can be equipped with displacement values (Oliveira *et al.*, 2000; Mantler *et al.*, 2007). Thus, although it is not new the idea of combining displacement mapping with texture lookups (Guskov *et al.*, 2000), the new features in Shader Model 4.0 offers us the opportunity of performing displacement mapping methods directly on the GPU and in a reliable way.

Gu *et al.* (2002) introduced Geometry Images, which capture geometry as a simple 2D array of quantized points. As opposed to remeshing an irregular mesh into one with a semi-regular connectivity, they proposed a technique to remesh an arbitrary surface onto a completely regular structure called a geometry image. With the introduction of faster texture fetchs in the shaders (common in the current vertex, geometry and fragment shaders) a geometry image can be rendered with high efficiency. However, this scheme does not provide level of detail features although improvements to this work introduced level of detail by means of mip mapping (Hernández and Rudomin, 2006).

On the other hand, according to Boubekeur and Schlick (2008), existing mesh refinement methods can be classified in direct and indirect refinement. Indirect refinement methods include pure geometry synthesis approaches, which mainly work in object-space, and direct ones, which are a kind of image processing algorithms (Bokeloh and Wand, 2006).

Some authors have employed precomputed pattern-based methods which need to transmit a coarse mesh through the BUS to be later refined on the GPU pipeline. These methods usually make use of vertex shaders (Ji *et al.*, 2005; Boubekeur and Schlick, 2005; Dyken *et al.*, 2008; Tatarchuk, 2008; Boubekeur and Schlick, 2008; Dyken *et al.*, 2009) or geometry shaders (Lorenz and Döllner, 2008) to emit these precalculated patterns for each coarse triangle of the mesh by copying it from vertex buffers. In general, memory footprint for these patterns is usually significant.

A recent method by Boubekeur and Schlick (2005), Boubekeur and Schlick (2008) proposed the idea of barycentric interpolation to perform refinement. By means of a vertex program, it replaces each coarse triangle by a precomputed tessellated triangle, which is then displaced according to a procedural function. However, it does not make use of geometry shader units to create geometry. Instead, tessellated triangles are all stored in video memory which noticeably increases its spatial cost and, moreover, the coarse mesh always needs to be transmitted through the BUS.

Table 1
Characterization of tessellation models.

Authors	Type	GPU usage	Coherence	Patterns complexity
Boubekeur and Schlick (2005)	Direct	Vertex	No	Very low
Bokeloh and Wand (2006)	Indirect	Vertex, pixel	Yes	–
Dyken *et al.* (2008)	Direct	Vertex, pixel	No	Medium
Lorenz and Döllner (2008)	Direct	Vertex, pixel, geometry	Yes	Medium-high
Boubekeur and Schlick (2008)	Direct	Vertex	No	High
Dyken *et al.* (2009),	Direct	Vertex, pixel	No	Medium
Our approach	Indirect	Vertex, pixel, geometry	Yes	Low

3.1. *Characterization of Tessellation Models*

Table 1 presents a comparison of the most recent methods from those considered in this review. The description takes into account the following aspects:

- Type: it indicates whether the refinement method is direct or indirect;
- GPU usage: it shows which GPU units are programmed;
- Coherence: it indicates whether the solution uses coherence to reuse calculations;
- Patterns complexity: it offers an estimation of the complexity of the tessellation patterns used. The complexity refers to the amount of patterns, their variability and the dependence to the model to be tessellated.

It is worth mentioning that our approach has also been included in this comparison. In this way, it can be seen how we have a low pattern complexity, how we exploit the latest graphics hardware and how we exploit coherence in contrast to many previous solutions. More precisely, we do not make use of precomputed patterns to refine a mesh. We use a fixed and uniform subdivision criterion to perform tessellation and a displacement map to compute the new vertices positions on the mesh. In brief, having initially computed the displacement map, we tessellate the mesh and, then, the provided (u, v) domain coordinates are used in the geometry shader to derive the actual vertices from the displacement map. Hence, a combination of tessellation and displacement mapping can be done by using the geometry shader stage and its amplification capabilities. As our approach mainly works in object-space, it falls into the category of indirect refinement methods (Boubekeur and Schlick, 2008).

Complexity and an efficient GPU usage have usually been a barrier when implementing rendering methods in standard 3D tools. Our method provides a low complexity but exploiting, at the same time, graphics hardware. It also provides an easy integration into creation tools.

4. Mesh Refinement

This Section introduces a new approach to render triangle meshes by exploiting the features of Shader Model 4.0. The workflow architecture that we use is presented in Fig. 4.

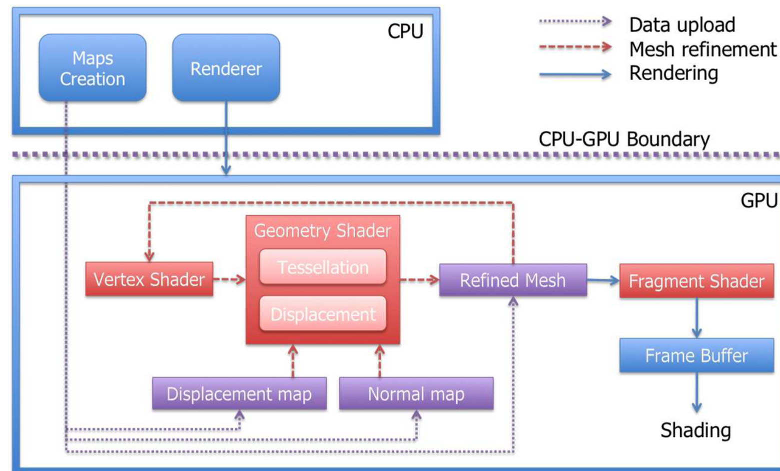

Fig. 4. Architecture of our mesh refinement approach.

As previously commented, we represent a surface as a coarse polygonal mesh plus a displacement function describing the difference between the original and the coarse mesh. Thus, we first pre-compute the coarse mesh and the displacement map. All these components are stored, once for all, on the GPU: coarse mesh as a vertex buffer object and the displacement map as a texture. In Fig. 4, they are represented as *Refined Mesh* and *Displacement map*, respectively. To clarify, we underline that *Refined Mesh* is employed in two contexts. On the one hand, it is used as input and output of the tessellation and displacement steps performed in the geometry shader. On the other hand, it is used as input to the fragment shaders for rendering on screen. Initially, the contents of the *Refined Mesh* correspond to the coarse mesh.

To create a differential displacement map, we need a pair of models: an original high detailed model and a simplified low-polygonal model or coarse model. We thus extract the differential displacement data by means of a technique to generate it. In our approach, we make use of a technique which exploits the precise information on models' vertex correspondence. In particular, ZBrush (Pixologic zbrush, `http://www.pixologic.com`) falls in this kind of techniques and is the software tool that we used in our examples. It provides a one-to-one correspondence between all the vertices of the simplified model and some of the vertices of the original model. Although there is no established standard for displacement maps, the most common representation is based on texture mapping approach, where each mesh vertex is bound with a pair [0, 1] range values which provides a mapping into the conventional (s, t) texture space of the displacement map, while the map itself is a two-dimensional image. As previously mentioned, during the initialization step, displacement is computed once for all and stored in video memory as a texture.

At rendering time, two successive operations are performed on the *Refined Mesh*: tessellation, for generating a refined mesh topology at the desired level of detail, and displacement, for translating each newly inserted vertex to its final position, obtained by sampling the continuous displacement function. More precisely, in the tessellation step we calculate

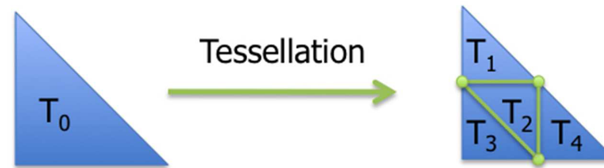

Fig. 5. Basic triangle operations performed in our approach. Edges forcing a split are marked with circles. One triangle split operation step means a subdivision in four new triangles. One triangle collapse operation means discard three triangles and amplify the other one to its original attributes.

an edge-based refinement for each triangle in a geometry shader and tessellate the triangle directly within the rendering pipeline. This can be done recursively by rendering the results of the geometry stage into a vertex buffer (*Refined Mesh*) and refeeding them into the pipeline (see Stream Output in Fig. 2). In the displacement step, we add small-scale geometric details by moving the vertex of the generated triangles along a vector provided by the displacement function. In our approach, the displacement of each vertex is constrained along its normal vector.

4.1. *Tessellation*

The tessellation step needs further explanation. The goal of the tessellation is to split triangles that do not approximate the input height map exactly enough into smaller ones. To maintain a continuous mesh topology at the same time, neighboring triangles must share the same tessellation along their common edge. As triangle-based refinement decisions can result in a differing refinement for the common edge of adjacent triangles, refinement decisions must be taken purely edge-based (Pulli and Segal, 1996) to guarantee a continuous topology.

Consequently, the first step to refine a triangle is to decide which of its edges need to be split. This refinement decision basically depends on the application, but predominately various error metrics can be applied. Possible error metrics include the height difference between an edge's spanning vertices as well as estimated object-space and screen-space errors between the coarse and the refined edge.

Depending on the result of the refinement decision, new vertices can be inserted along each respective edge. As we aim at uniform refinement of meshes, we always produce the same level of tessellation in the whole mesh by inserting one vertex at the midpoint of each edge. These decisions results in only one possible tessellation for each triangle (see Fig. 5). There are several reasons for not inserting more than one vertex along each edge:

- Simplicity: the most important reason is that one case of tessellation can be quickly handled. Increasing the number of splits yields in different cases for each triangle, including complicated refinement configurations.
- Expensive conditionals: more cases of tessellation require more conditional statements within the geometry shader, which are still very time-consuming on current graphics hardware.

- Geometry shader output limit: the output of the geometry shader stage is currently limited to 1,024 floats per call. With one split per triangle edge, this limit is never exceeded, avoiding problematic cases of incomplete refinements.

4.2. *Coherence*

Coherence means taking advantage of the last information obtained in the refinement of the mesh. The use of coherence reduces the time spent by the mesh refinement process on avoiding the repetition of tessellations that have already been performed. As a result, the refined mesh always represents the current tessellated mesh and, contrary to other approaches (Boubekeur and Schlick, 2005; Boubekeur and Schlick, 2008; Dyken *et al.*, 2009), we are able to go forward and backwards in the refined mesh without the need of starting from the coarse mesh every frame and refine it until a target refined mesh resolution is achieved. In this way, we take advantage of coherence in transitions between different refined meshes.

Until now, we have only discussed one tessellation step. However our approach is recursive and it increases the mesh resolution iteratively. Shader programs however, cannot use recursion, not even statically. The only way to apply a shader recursively is to employ multiple render passes, only asserting that the data format does not change in between the calls.

Therefore, the output of a geometry shader pass must be captured in a buffer to be reused in a consecutive pass. A technique that provides this functionality for OpenGL is transform feedback (Barthold and Brown, 2008). To increase performance, the rendering pipeline can be aborted after the application of the geometry shader if more than one pass is scheduled.

The special advantage of this course of action is that the fragment stage needs to be passed only once even if the approach employs several rendering passes. This becomes very important when complex fragment shaders are applied to add visual details, as their costs remain constant with this proceeding.

Only two pieces of information need to be stored in the transform feedback buffer: the vertex positions and the texture coordinates. The vertex positions are interpolated while tessellating triangles and the texture coordinates are used to interpolate height, normal and texture values for new vertices from the corresponding maps.

After a rendering pass is complete, the transform feedback buffer is bound as a vertex buffer for the next rendering pass. To allow for more than one pass, a second buffer is used as transform feedback buffer for the second pass. These two buffers alternately operate as source vertex buffer and target transform feedback buffer. More than these two buffers are never required.

The described procedure is used as a dynamic level of detail algorithm. As our approach purely operates on the GPU, the CPU merely has to start it each frame and can then concentrate on other calculations. If the CPU regularly checks whether the transform feedback buffer is ready and there is enough time left for the current frame, it can reapply the refinement process to the result of the previous frame to further improve mesh detail.

Alternatively to this refinement process with coherence, we also could perform a typical refinement method by always starting from a coarse mesh and obtain a deeper refinement level with a multi-pass geometry shader rendering. This alternative method is also analyzed in the Section 5.

4.3. *Rendering*

It is important to underline that the coarse mesh uploaded to the *Refined Mesh* buffer is initialized with the height values from the displacement map before it enters the recursion loop. This has the positive effect that the shader can perform faster as it has to conduct texture lookups only for newly generated vertices as opposed to looking up a height value for every output vertex.

Algorithm 1 CPU-side rendering loop

```
// Refinement step
if newRefinementLevel() then
  if lastVBOused is Even then
    BindVertexBuffer(VB1)
    RenderToVertexBuffer(VB0)
    lastVBOused ← 1
  else
    BindVertexBuffer(VB0)
    RenderToVertexBuffer(VB1)
    lastVBOused ← 0
  end if
end if

// Render Refined Mesh
if lastVBOused is Even then
  RenderToScreen(VB1)
else
  RenderToScreen(VB0)
end if
```

Algorithm 1 shows the basic pseudo-code associated to CPU-side rendering loop. The *newRefinementLevel* function determines if a new refinement level has been reached on the GPU. Such a situation could be computed by comparing the output triangle count of the geometry shader from two consecutive passes. If the difference is too small, no new refinement is needed and the mesh is directly rendered. Another method could also be to control if the mesh resolution exceeds the displacement map resolution which would produce over-refined triangles scenarios.

After the pass on the GPU has been computed, the mesh is rendered and the resulting mesh is stored in the corresponding vertex buffer. We alternately render the contents of

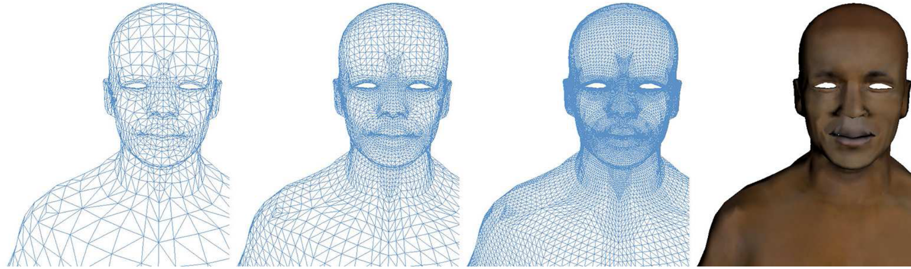

Fig. 6. Mesh refinement of the senna mesh. From left to right, coarse mesh with 6,364 triangles, and mesh refinements with 25,456, 101,824 and 407,296 triangles.

one vertex buffer and store the results in the other one. This way, it never needs more than two buffers, no matter how many loops are actually passed. Finally, we render the resulting mesh to the screen. In this way, when no refinements are required, we obtain rendering performances equal to the one obtained with static meshes.

As previously mentioned, the essential workflow of our algorithm in the graphics pipeline is illustrated in Fig. 4. In each call, the geometry shader first checks if the triangle needs refinement or simplification by means of the *CalculateRefinementLevel* function. This method determines the recursion depth of the algorithm. This can be a fixed number or a dynamically calculated one. That depends on situational circumstances such as the viewer's distance to the object. Furthermore, the refinement step can be avoided if no additional detail is needed. Thus, the results of this function can be to refine, to simplify or no operation with the current *refined* mesh, avoiding degenerate situations such as one-pixel micro-polygons or stripe-shaped triangles.

On the one hand, refinement operation computes three new vertices and emits four new triangles. Tessellation decisions are discussed in Section 4.1. On the other hand, simplification is performed by discarding some triangles and modifying others. Finally, the *OutputTriangles* method emits the triangle or triangles to the corresponding vertex buffer bound.

5. Experimental Results and Discussion

All results were obtained on a Mac Pro 2.8 GHz with 4 GB of RAM and with an NVidia Geforce 8800 GT video card with 512 MB of memory at a view port of 800×600. Implementation was performed in GLSL as the shader programming language; OpenGL was used as the supporting graphics library and C++.

In Fig. 6, we present the evolution in the refinement of the senna mesh. The coarse mesh, located in GPU memory, is composed of 6 364 triangles. At runtime, this mesh is progressively refined in real-time by using our approach. We have used a single heightmap to displace the refined tessellation. Regarding memory usage, we have no memory overhead on the CPU side as all data are located on GPU. On GPU side, we store the coarse mesh plus the displacement and texture map. In the case of the senna mesh, we uploaded 6 364 triangles and a displacement map of $4\,096 \times 4\,096$ which offers us a maximum tessellation level of up to 8M triangles at a cost of around 8 MB in video memory. Moreover,

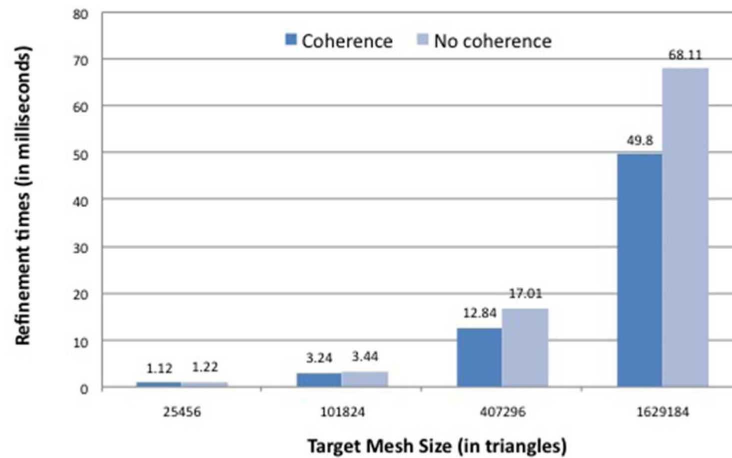

Fig. 7. Comparison between a coherence and a non-coherence approach. For the largest target mesh size (1.6M triangles), the coherence method offers a performance that is around 35% faster than the no coherence refinement solution.

we compared our approach with a well-know multiresolution model: Progressive Meshes (Hoppe, 1996). From version 5.0, it has been included in the Microsoft DirectX graphics library. We can observe an improvement of around four times in the speed between using our approach and Progressive Meshes.

Figure 7 presents the times required in the graphics pipeline to perform two types of refinements on the same model. Refinement with coherence and with no coherence (see Section 4.2). In brief, coherence always maintains the current refined mesh in memory and it does not need to start from a coarse mesh and refine it to a target mesh size, which is the case of the no coherence refinement. Thus, while providing the same final image quality, we can observe a gain by comparing both types of refinement of around 35 per cent of time with a target mesh size of 1.6M triangles. This can explained by a more intensive use of the geometry shader, that is, refinement with coherence do not need as many passes as the no coherence solutions.

Finally, Fig. 8 shows the rendering frame rate obtained for the senna model for different target mesh sizes. The measure integrates the tessellation and displacement step. It clearly appears that refinement does not have a high influence in the performance of the GPU, that is, rendering times are more limited by the GPU capabilities than by our refinement method. Moreover, by using coherence, we always fit into memory the number of triangles to be rendered and we do not need to always transmit a coarse mesh from the CPU which becomes a bottleneck on the graphics BUS in most of the precomputed pattern-based approaches.

6. Conclusions and Future Work

We have presented a method for geometry synthesis by mesh refinement that efficiently combines new GPUs capabilities and displacement mapping. As it enables us to increase

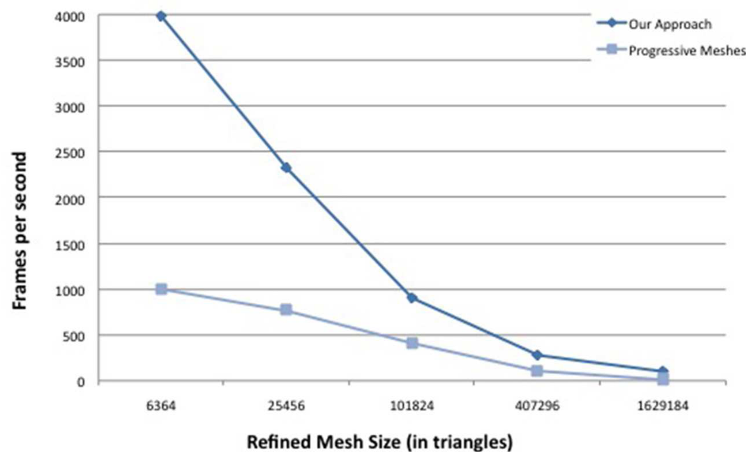

Fig. 8. Frame rate measures for a linear transition among different mesh resolutions.

detail independently from CPU, it is suitable for applications that requires a huge refinement depth, such as CAD or scientific visualization. The CPU processing is reduced to select the appropriate buffer with no transmission of any mesh to the GPU at rendering time.

Our method offers a flexible integration with standard 3D tools, such as Pixologic ZBrush or 3D Studio Max, that makes it easier to be implemented in final applications. Moreover, it goes further on GPU geometric processing, since it consistently performs geometry synthesis in object space. It also offers a flexible and efficient method for non-precomputed pattern-based approaches. In contrast to these approaches, we are able to render a refined mesh almost independently of the amount of available memory (the only limitation is the storage on GPU of the maps).

As future work, we would like to explore adaptive refinement by using our method. By offering an edge-based tessellation decision instead of a triangle-based, we would obtain more precise tessellations. However, new possible triangulations should be applied.

Another interesting line is the appearance of a more general and optimized tessellation support that will be introduced by future hardware for the upcoming Direct3D 11 (Gee, 2008), which adds three more pipeline stages (hull shader, tessellator, and domain shader).

Acknowledgements. This work was supported by the Spanish Ministry of Science and Technology (Project TIN2010-21089-C03-03) and Bancaixa (Project P1.1B2012-40).

References

Asirvatham, A., Hoppe, H. (2005). Terrain rendering using GPU-based geometry clipmaps. In: *GPU Gems 2*, pp. 26–43.

Barthold, L., Brown, P., W.E. (2008). Documentation for opengl nv_transform_feedback extension. `http://www.opengl.org/registry/specs/nv/transform_feedback.txt`.

Blasco, O. (2002). Curvature simulation using normal maps. In: *Game Programming Gems 3*.

Blythe, D. (2006). The direct3D 10 system. *ACM Transactions on Graphics*, 25(3), 724–734. http://doi.acm.org/10.1145/1141911.1141947.

Bokeloh, M., Wand, M. (2006). Hardware accelerated multi-resolution geometry synthesis. In: *I3D'06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pp. 191–198.

Boubekeur, T., Schlick, C. (2005). Generic mesh refinement on GPU. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '05)*. ACM, New York, pp. 99–104. http://doi.acm.org/10.1145/1071866.1071882.

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

Catmull, E.E. (1974). *A subdivision algorithm for computer display of curved surfaces*. PhD Thesis, The University of Utah.

Cignoni, P., Montani, C., Scopigno, R., Rocchini, C. (1998). A general method for preserving attribute values on simplified meshes. In: *Proceedings of the conference on Visualization'98 (VIS'98)*, pp. 59–66.

Doggett, M., Hirche, J. (2000). Adaptive view dependent tessellation of displacement maps. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS'00)*, pp. 59–66.

Doggett, M.C., Kugler, A., Strasser, W. (2001). Displacement mapping using scan conversion hardware architectures. *Computer Graphics Forum*, 20(1), 13–26.

Dyken, C., Reimers, M., Seland, J. (2008). Real-time GPU silhouette refinement using adaptively blended Bezier patches. *Computer Graphics Forum*, 27(1), 1–12.

Dyken, C., Reimers, M., Seland, J. (2009). Semi-uniform adaptive patch tessellation. *Computer Graphics Forum*, 28(8), 2255–2263.

Espino, F.J., Boo, M., Amor, M., Bruguera, J.D. (2005). Adaptive tessellation of Bezier surfaces based on displacement maps. In: *Proc. 13th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG05)*, Plzen, Czech Republic, pp. 29–32.

Gee, K. (2008). Direct3D 11 tessellation. *Presentation Gamefest.* http://www.microsoft.com/downloads/details.aspx?FamilyID=2d5bc492-0e5c-4317-8170-e952dca10d46.

Gu, X., Gortler, S.J., Hoppe, H. (2002). Geometry images. In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press, New York, pp. 355–361.

Gumhold, S., Hüttner, T. (1999). Multiresolution rendering with displacement mapping. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware (HWWS'99)*, pp. 55–66.

Guskov, I., Vidimce, K., Sweldens, W., Schroder, P. (2000). Normal meshes. In: *ACM SIGGRAPH*, pp. 95–102.

Hernández, B., Rudomin, I. (2006). Simple dynamic lod for geometry images. In: *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia (GRAPHITE'06)*, pp. 157–163.

Hoppe, H. (1966). Progressive meshes. In: *Proceedings of SIGGRAPH, ACM SIGGRAPH'06*, pp. 99–108.

Jeschke, S., Wimmer, M., Purgathofer, W. (2005). Image-based representations for accelerated rendering of complex scenes. In: *EUROGRAPHICS 2005 State of the Art Reports*, pp. 1–20.

Ji, J., Wu, E., Li, S., Liu, X. (2005). Dynamic lod on GPU. In: *Computer Graphics International 2005*, pp. 108–114.

Lorenz, H., Döllner, J. (2008). Dynamic mesh refinement on GPU using geometry shaders. In: *Proceedings of the 16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2008*, pp. 97–104.

Mantler, S., Jeschke, S., Wimmer, M. (2007). Displacement mapped billboard clouds. In: *Symposium on Interactive 3D Graphics and Games*.

Moule, K., Mccool, M.D. (2002). Efficient bounded adaptive tessellation of displacement maps. In: *Graphics Interface*, pp. 171–180.

Oliveira, M.M., Bishop, G., McAllister, D. (2000). Relief texture mapping. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'00)*, pp. 359–368.

Pulli, K., Segal, M. (1996). Fast rendering of subdivision surfaces. In: *Proceedings of the Eurographics Workshop on Rendering Techniques'96*. Springer, London, pp. 61–70.

Szirmay-Kalos, L., Umenhoffer, T. (2008). Displacement mapping on the GPU – State of the Art. *Computer Graphics Forum*, 27(6), 1567–1592.

Tatarchuk, N. (2008). *Advanced topics in GPU tessellation: algorithms and lessons learned*. Presentation, Gamefest. http://developer.amd.com /gpu_assets/Tatarchuk-Tessellation (Gamefest2008).pdf.

F. Ramos is an associate professor in the Department of Computer Languages and Systems at the University Jaume I of Castellon. He got his bachelor and master degrees in computer science from this University. He got his PhD from University Jaume I of Castellon in 2008. His research interests are in the areas of computer graphics, geometric modeling and visualization.

M. Chover received his MS degree in Computer Science in 1992, and his PhD in Computer Science in 1996, from the Universidad Politecnica de Valencia, Valencia, Spain. Since 1992, he has been an Assistant Professor of Computer Science at the department of Computer Languages and Systems at the Universitat Jaume I, Spain. He is member of the executive committee of Eurographics (Spanish Chapter). His research areas include multiresolution modelling, real-time visualization, collaborative virtual worlds and virtual and augmented Reality.

O. Ripolles received his degree in Computer Engineering in 2004 and his PhD in 2009 at the Universitat Jaume I in Castellon (Spain). He has also been a researcher at the Université de Limoges (France) and at the Universidad Politecnica de Valencia (Spain). He is currently woking in neuroimaging at Neuroelectrics in Barcelona (Spain). His research interests include multiresolution modeling, geometry optimization, hardware programming and medical imaging.

Vaizdų skyros analizė 3D modelių atvaizdavimui

Francisco RAMOS, Miguel CHOVER, Oscar RIPOLLES

Vaizdų sintezės metodai yra pritaikomi daugybėje sričių, kadangi sumažina reikiamą informacijos kiekį kuriant realistines vizualizacijas. Aparatūriniame lygmenyje atvaizdavimas vyksta vaizduojamo objekto geometrijos aprašymus trikampiais. Šiame straipsnyje autoriai pristato naują, paprastą karkasą skirtą realiu laiku patikslinti ir supaprastinti trikampių tinklą, skaičiavimus atliekant grafiniame procesoriuje. Tikslui pasiekti reikalingas pradinis tinklas, pakeitimų žemėlapis ir informacija apie atvaizduojamo objekto sudaromus šešėlius. Atvaizdavimo etape objekto sudaromi šešėliai yra panaudojami išlygiagretinant mozaikos sudarymą ir pakeitimų žemėlapio perskaičiavimui. Siūlomas metodas atsižvelgia į prieš tai buvusioje iteracijoje gautus rezultatus.