Real-time Tessellation of Terrain on Graphics Hardware

Oscar Ripolles *,

Universitat Politecnica de Valencia, Valencia, Spain

Francisco Ramos, Anna Puig-Centelles and Miguel Chover

Universitat Jaume I, Castellón, Spain

Abstract

Synthetic terrain is a key element in many applications that can lessen the sense of realism if it is not handled correctly. We propose a new technique for visualizing terrain surfaces by tessellating them on the GPU. The presented algorithm introduces a new adaptive tessellation scheme for managing the level of detail of the terrain mesh, avoiding the appearance of t-vertices that can produce visually disturbing artifacts. Previous solutions exploited the Geometry Shader capabilities to tessellate meshes from scratch. In contrast, we reuse the already calculated data to minimise the operations performed in the shader units. These features allow us to increase performance through smart refining and coarsening.

Key words: Terrain simulation, Tessellation, Level of Detail, Real-time Rendering, GPU

* Corresponding author. Tel.: +34 963877000 ext. 88442
Email addresses: oripolles@ai2.upv.es (Oscar Ripolles),

1 Introduction

In recent years the research area of procedural modeling has been the focus of a lot of effort. The latest works try to take advantage of the new graphics hardware technology, making it possible for the geometry to be generated at rendering time in the graphics card itself. Thus, instead of specifying the details of a 3D object, we provide some parameters for a procedure that will create the object.

In the field of computer graphics, tessellation techniques are often used to divide a surface into a set of polygons. Thus, we can tessellate a polygon and convert it into a set of triangles or we can tessellate a curved surface. These approaches are typically used to amplify coarse geometry. Programmable graphics hardware has enabled many surface tessellation approaches to be migrated to the GPU, including isosurface extraction (Buatois et al., 2006), subdivison surfaces (Shiue et al., 2005), NURBS patches (Guthe et al., 2005), and procedural detail (Bokeloh and Wand, 2006; Boubekeur and Schlick, 2005). In this paper we analyse the possibilities offered by GPU-based tessellation techniques for terrain visualisation.

For many decades terrain simulation has been the subject of research, and there are many solutions in the literature to its realistic and interactive rendering. Most of these solutions simulate terrain as an unbounded surface that is represented in the synthetic environment as a heightmap, which is a regularly-spaced two-dimensional grid of height coordinates. These grids can be later processed by a modeling software or a rendering engine to obtain the 3D sur-

francisco.ramos@uji.es (Francisco Ramos), apuig@uji.es (Anna Puig-Centelles), chover@uji.es (Miguel Chover).

face of the desired terrain.

Some authors have criticised the use of heightfields, as these data structures store only one height value for any given (x,y) pair. In this sense, in specific cases like caves or complex terrain formations it may be possible to have more than one height value for each position. In our case we will not consider these complex formations and, thus, the use of a squared heightmap will still be adequate.

We introduce a new adaptive tessellation scheme for terrain that works completely on the GPU. The main feature of the framework that we are presenting is the possibility of refining or coarsening the mesh while maintaining *coherence*. By coherence we refer to the reuse of information between changes in the level of detail. In this way, the latest approximation extracted is used in the next step, optimising the tessellation process and improving performance.

The rest of the paper is structured as follows. Section 2 presents the state of the art in terrain simulation. Section 3 thoroughly describes our tessellation technique. Section 4 offers some results on the technique presented and, lastly, Section 5 presents some conclusions on the techniques developed and outlines future work.

2 Related work

Digital Terrain Models (DTMs) are usually represented and managed by means of regular or irregular grids. The reader is referred to recent surveys for a more in-depth review of these methods (Pajarola and Gobbetti, 2007; Rebollo et al., 2004).

2.1 Regular Grids

The most common regular structures are quadtrees and binary trees (bintrees). These structures with regular connectivity are suitable for terrain, as the input data usually come as a grid of values. In this sense, regular approaches have produced some of the most efficient systems to date (Pajarola and Gobbetti, 2007).

Quadtrees are found in the literature in many papers with CPU-based solutions (Lindstrom et al., 1996; Pajarola, 1998) as well as GPU-based approximations (Schmiade, 2008). This latter approach proposed a tessellation algorithm on the GPU, although the pattern selection process was very complex.

The ROAM method (Real-time Optimally Adapting Meshes) (Duchaineau et al., 1997) is a widely known method based on the use of bintrees. They use two priority queues to manage split and merge operations, obtaining high accuracy and performance. As an attempt to improve this solution, in (Apu and Gavrilova, 2004) the authors eliminated the priority queue for merges to exploit frame-to-frame coherence.

Some authors proposed using bintrees where each node contains, instead of a single triangle, a patch of triangles (Levenberg, 2002; Pomeranz, 2000). Algorithms like (Cignoni et al., 2003; Schneider and Westermann, 2006) batched the triangular patches to the graphics hardware. The Batched Dynamic Adaptive Mesh (BDAM) proposed in (Cignoni et al., 2003) used triangle strips to increase performance, although it was based on complex data structures and costly processes which still produced popping artifacts. From a different perspective, (Larsen and Christensen, 2003) used patches of quads to manage terrain rendering on the GPU. Later, (Schneider and Westermann, 2006) reduced bandwidth requirements by simplifying the mesh and using progressive transmission of geometry. Recently, in (Bosch et al., 2009) the authors proposed the use of precalculated triangle patches to develop a GPU-intensive solution.

The projected grid concept offered an alternative way to render displaced surfaces with high efficiency (Johanson, 2004). The idea was to create a grid with vertices that were evenly-spaced in post-perspective camera space. This representation provided spatial scalability and a fully GPU-based implementation was described. In (Schneider et al., 2006), Schneider et al. used the projective grid method to render infinite terrain in high detail. They generated the terrain in real time on the GPU by means of fractals. The work in (Livny et al., 2008) proposed using ray tracing to guide the sampling of the terrain, being the technique able to produce both regular and irregular meshes.

As an improvement over binary trees, (Losasso and Hoppe, 2004) introduced *geometry clipmaps*, caching the terrain in a set of nested regular grids. These grids were stored as vertex buffers, and mipmapping was applied to prevent aliasing. As vertex buffers cannot be modified on the GPU, this approach was later improved by using vertex textures to avoid having to use the CPU to modify the grids (Asirvatham and Hoppe, 2005). This work was also extended to handle spherical terrains (Clasen and Hege, 2006).

Lastly, it is worth mentioning that bintree hierarchies are also useful for decompressing terrain surfaces on the GPU. In this sense, (Lindstrom and Cohen, 2010) presented a fast, lossless compression codec for terrains on the GPU, and demonstrated its use for interactive visualisation.

2.2 Irregular Grids

Irregular grids are commonly known as TINs (Triangulated Irregular Networks) and represent surfaces through a polyhedron with triangular faces. These solutions are less constrained triangulations of the terrain and, in general, need fewer triangles although their algorithms and data structures tend to be more complex.

(Hoppe, 1998) proposed specialising his View-dependent Progressive Mesh (VDPM) framework for arbitrary meshes that represent terrain. With a more intense GPU exploitation, (Dachsbacher and Stamminger, 2004) proposed a costly procedural solution that needs three rendering passes to obtain the geometry.

Delaunay triangulation is one of the main techniques used to create the terrain mesh. In computational geometry, a Delaunay triangulation for a set of points is a triangulation where no point is inside the circumcircle (circle which passes through all the vertices of the triangle) of any generated triangle (Delaunay, 1934). This triangulation has been widely used in terrain solutions (De Berg et al., 2008; Rabinovich and Gotsman, 1997). The main problem with Delaunay triangulation is that it relies on smoothing morphing between two triangulations generated in two successive frames, but the triangulations may be very different. As an improvement, (Cohen-Or and Levanoni, 1996) proposed a solution that involved blending between two levels of Delaunay triangulation without adding more triangles. More recently, (Liu et al., 2010) proposed a new technique where points from a DEM are initially given an importance value in order to guide the adaptive triangulation in real time. Their proposal allowed for smooth morphing and tried to eliminate very small triangles which could produce visual disturbances.

As a conclusion, we can note that techniques based on irregular grids tend to be more complex and less suitable for GPU computations.

3 Our GPU-based Tessellation Scheme

In this paper we propose a new adaptive tessellation algorithm that works completely on the GPU. Moreover, this algorithm is able to offer view-dependent approximations where more detail is added in areas of interest. Our algorithm will be based on bintrees, creating the hierarchy on the GPU using some specific equations. As mentioned above, there have been different proposals with similar aims, although our scheme is easier to implement while still robust and efficient.

A successful tessellation algorithm is based on the selection of the most suitable tessellation patterns to amplify the triangles. These patterns affect the algorithm chosen to refine and coarsen the geometry. As our aim is to process the mesh in a *Geometry Shader*, each triangle is to be processed separately and in parallel. Thus, it will be necessary to develop a technique to alter the geometry of the different triangles without any communication between them. Moreover, the algorithms must assure that no cracks or holes appear on the surface mesh.

In the remainder of this section we will address the selection of the patterns and also the algorithms to manage the terrain surface.

3.1 Tessellation patterns

It is possible to find in the literature different proposals of tessellation patterns. Among them, we have selected the seven patterns presented in (Schmiade, 2008). Figure 1 presents, on the left side, an initial rectangular triangle where its hypotenuse and catheti (or *legs*) are labeled as H, C_1 and C_2 respectively. Next, the seven tessellation patterns are presented, where the edges of the original triangle that need refinement are depicted in red.

These patterns assure that no *t-vertices* are produced. A *t-vertex* appears after a tessellation step when two edge junctions make a *t-shape* (McConnell, 2006). To clarify the appearance of *t-vertices*, Figure 2 presents the initial geometry of a terrain composed of three triangles. According to some criterion, it is decided to refine the middle triangle and vertex v_5 is introduced, outputting 4 triangles. Then, if we decided to apply a heightmap to this geometry, we would find holes around vertex v_5, as the triangle on top has no reference to this vertex. In the example offered, vertex v_5 would be a *t-vertex*.

The different patterns presented in Figure 1 offer a robust tessellation and avoid cracks and holes. We must note that our tessellation algorithm is based on the use of an edge-based criterion to decide which triangles to refine and which to coarsen. This is an improvement over triangle-based criterions, as they tend to introduce *t-vertices*. In this sense, each pattern shows the tessellation that would be necessary depending on the combination of edges that need refinement. We use the center point of each edge to perform the appropriate calculations. For example, pattern 3 considers a situation in which the hypotenuse needed refinement and a new vertex has been added to create two new triangles.

In addition to the edge-based criterion, we also must select a metric to decide whether an edge should be refined or coarsened. Although in most cases the distance to the camera is the selected metric, it would be possible to apply more accurate heuristics that balance the perceived visual quality and the extraction time that the tessellation process needs.

3.2 Tessellation algorithm

At each iteration of the tessellation process, the algorithm checks each triangle to see whether it is necessary to refine or coarsen it. More precisely, the algorithm checks the center point of each edge according to the selected metric. The resulting combination of edges that need processing indicates which pattern should be applied.

For the correct performance of our tessellation scheme, it is necessary for each triangle to store:

- The spatial, texture and normal coordinates.
- A number indicating the id of the triangle.
- A number coding the tessellation patterns that have been applied ($patternInfo$).

The need of storing the id and $patternInfo$ values is due to the fact that we must know how a particular triangle was created in order to know how we should modify it when swapping to a lower level of detail. This information is crucial as it allows the algorithm to coarsen the geometry without having to return to the initial mesh. This is one of the main contributions of our algorithm, as coherence among extracted approximations considerably increases the rendering performance.

On the one hand, the id value will uniquely identify each of the generated triangles and will also allow us to calculate the id of its parent triangle. This id number is calculated following the formula:

$$id = id * maxOutput + originalTris + childType \tag{1}$$

The $maxOutput$ value is understood as the maximum number of triangles that can be output from a parent triangle using the available patterns. The patterns presented in Figure 1 involve outputting a maximum number of four new triangles and, as a consequence, in our case $maxOutput$ is a value equal to 4. The $originalTris$ constant refers to the number of initially existing triangles on the source mesh, which depends on the input mesh the application uses. Finally, $childType$ is a value used to differentiate between the triangles output from a parent triangle. As the patterns output a maximum number of four new triangles, in our case the $childType$ is a value ranging from 0 to 3. The $childType$ value assures that each id is different, allowing us to distinguish between the triangles that belong to the same parent. This feature is compulsory for the correct simplification of the mesh.

On the other hand, the $patternInfo$ number is used to store all the patterns that have been applied to refine a triangle. Equation 2 has been specifically prepared to code the different patterns applied to a triangle in one single value. In this equation, $latestPattern$ refers to the type of the latest pattern, the one that we have used to create this triangle. The value $numberOfPatterns$ refers to the number of available patterns that we can apply in our tessellation algorithm. In our case we use the seven patterns presented in Figure 1 and, as a consequence, $numberOfPatterns$ should be equal to 7. It is worth mentioning that, initially, all the $patternInfo$ values are equal to 0. This $patternInfo$

value will be the same for all the triangles belonging to the same parent. This piece of information is important to know how a particular triangle was created and, consequently, how we should modify it when swapping to a lower level of detail.

$$patternInfo = patternInfo * numberOfPatterns + latestPattern \quad (2)$$

Refining algorithm

When *refining* the mesh, the algorithm checks the center point of each edge to see whether they need refinement. Depending on the combination of edges that need more detail, the algorithm selects a tessellation pattern (see Figure 1) and generates the adequate number of triangles. For each new triangle, the algorithm calculates its spatial coordinates, texture information and any other information needed for rendering. To clarify the process, Figure 3 presents an example of how the tessellation process works. We present the initial mesh composed of three triangles, initially labeled with *id*s 0, 1 and 2. The dotted line represents the plane that we will use to define which area of the mesh needs refinements, being the area on the left the one that requires more detail. Each of the initial triangles goes through the extraction process of the algorithm that we are presenting.

In the specific case of triangle number 2, the algorithm detects that none of its edges needs refinement and, as a consequence, no change will be made. Nevertheless, the algorithm detects that triangle with *id* 0 needs refinement because the center point of some of its edges is on the left of the dotted line. Then, the algorithm chooses pattern 6 as it reflects the combination of edges to be refined. Using this pattern, the tessellation process algorithm generates the three new triangles shown in the figure. It can be seen how the id values of the new triangles are calculated following the formula 1, assuring that no repeated id is given. The triangle with id 1 is similarly refined using pattern 6.

Following on with the refinement process, the next tessellation step shows that different patterns have been applied to obtain different types of tessellation. In the figure we depict how triangle 5 is refined with pattern 2 and triangle 7 with pattern 7. It is important to mention that this figure also includes the $patternInfo$ of the different triangles, which is calculated using Equation 2.

Coarsening algorithm

A different process should be applied when diminishing the detail of the mesh. The id and the $patternInfo$ values of the triangles have been precisely given in order to simplify the coarsening process. Using the example given in the previous subsection, let us suppose that we want to reduce the detail and return to the state shown in the middle of Figure 3. In this case, each of the triangles located on the left of the dotted line would execute the same coarsening process.

When tessellating a triangle, for example with the pattern that is used when the hypotenuse and both legs are refined (see pattern 7 in Figure 1), four triangles are output. Nevertheless, only one of these triangles will be needed when diminishing the detail. As we will see in the remaining of this Section, three of them will be discarded and the other one will be modified to recreate the geometry of the parent triangle.

When coarsening the mesh, the first step is to find out whether the triangle that we are processing can be discarded or if it is the triangle in charge of retrieving the geometry of the parent triangle. The $childType$ used when calculating the id of each triangle is necessary for this particular differentiation. In those cases where this value is equal to 0, the algorithm assumes this triangle is in charge of recovering the geometry of the parent triangle; if the value is not equal to 0, the triangle is discarded. The $childType$ can be retrieved by using Equation 3.

$$childType = mod((id - originalTris), maxOutput) \tag{3}$$

$$id = (id - originalTris)/maxOutput \tag{4}$$

The second step entails knowing which pattern was applied to create the existing triangle. This is due to the fact that for each pattern we will perform different calculations for retrieving the three vertices of the parent triangle. In this situation, the $patternInfo$ value helps us to know which pattern was applied, as the latest pattern can be obtained with the next equation:

$$latestPattern = mod(patternInfo, numberOfPatterns) \tag{5}$$

Once we know which pattern was applied, we calculate the position of the vertices and we output the new geometry with the new id value obtained in Equation 4 and the new $patternInfo$ value obtained with Equation 6. The way we calculate these values assures that we will be able to continue coarsening the mesh or refining it without any problem.

$$patternInfo = patternInfo/numberOfPatterns \tag{6}$$

Following on with the example presented in Figure 3, let us suppose that we are processing the triangle with id 34. If we calculate its $childType$ we obtain a value greater than 0, indicating that it can be discarded. Nevertheless, triangle 31 has a $childType$ value equal to 0 and, thus, it is the one used to recover the parent triangle. The id of the parent triangle can be obtained with Equation 4. In this case, the $latestPattern$ would indicate that pattern 7 was applied and we would calculate the spatial coordinates of the parent triangle accordingly. Once again we would like to remember that all these operations have been coded in the shaders, so that the algorithm knows which operations to perform depending on the type of pattern applied.

Global algorithm

Once we have described the main characteristics of our algorithm, we must consider how the refining and coarsening processes work together. The refinement process is executed at each frame while the criterion is met and until we reach a maximum tessellation level, which is defined by the application. Similarly, the coarsening process is performed at each frame until the original geometry is obtained.

Nevertheless, in a real application the surface representing the terrain is refined and coarsened at the same time, as the tessellation conditions are modified while the user navigates through the scene. Our algorithm is capable of handling multiple levels of detail on the mesh, as the tessellation is applied to each edge of the triangles individually. In this sense, it is possible that in the triangle some edges need refinement and some need simplification. In these cases, and as it happened in the examples above, the algorithm would choose the most suitable pattern that fits this situation.

We must note that we will not store precomputed patterns on GPU memory as other solutions do (Boubekeur and Schlick, 2005). We just code in the *Geometry Shader* the seven cases that we follow (see Figure 1)so that the coordinates of the new vertices can be calculated from the coordinates of existing vertices when refining and coarsening the triangles.

Finally, it is worth mentioning that the last step of our algorithm includes retrieving the height of the newly computed vertices from the heightmap, which is previously stored in the GPU. It can be seen as the use of a displacement map to alter the position of each vertex (Szirmay-Kalos and Umenhoffer, 2008).

In Figure 4 we present an overview of the management of the heightmaps. First, the whole heightmap is allocated into main memory. Before the rendering stage starts, the area to be initially processed is uploaded to graphics memory. When the area of interest changes, a new texture should be uploaded to GPU memory. In order to avoid GPU stalls during these texture streamings, our approach uses asynchronous updates by means of Pixel Buffer Objects. A Pixel Buffer Object Elhassan (2005) is simply an array of bytes in GPU memory. However, this type of object can improve performance because it allows the graphics driver to streamline writing to video memory and to schedule asynchronous transfers. Thus, CPU does not need to wait for the texture transfer to be completed. In Figure 5, we present a graphical comparison between the conventional manner to load a texture from main memory to graphics memory and the alternative method by using a Pixel Buffer Object. The conventional method requires the CPU to perform all the transfer processes. On the contrary, with the PBO, the CPU still has to perform the transfer of data, but transferring data from the PBO to the texture object is managed by the GPU. Therefore, OpenGL performs these transfer operations without the CPU intervention and asynchronous operations in memory can be scheduled while rendering.

The texture upload could be triggered when the user approaches one of the limits of the terrain. When this situation happens, the systems starts streaming the new heightmap to the PBO, replacing those areas which are no longer used. At the same time,

4 Results

In this section we will study the performance of our tessellation method by analysing the visual quality obtained as well as the calculation time of the extracted approximation. Our scheme was programmed with GLSL and C++ on a Windows Vista Operating System. The tests were carried out on a Pentium D 2.8 GHz. with 2 GB. RAM and an nVidia GeForce 8800 GT graphics card.

4.1 Visual Results

First, we offer some visual results of the tessellation algorithm that we have described. Figures 6 and 7 present a mesh in wireframe where different tessellations have been applied. These figures show how the tessellation process is capable of increasing the detail of an input mesh without introducing cracks or other artifacts.

Figure 6 presents a tessellation case where an initial mesh (on top) is refined according to the distance to the camera. In this Figure the height values are recovered from a heightmap stored as a texture on the GPU. We have also included an image of the texturing process that can also be applied in our process, as the algorithm can calculate the texture coordinates when tessellating the surface mesh.

We can find another tessellation example in Figure 7 where five tessellation steps are presented. In this case, we have considered that a fictitious frustum has been located on the mesh to guide the tessellation process which considers the distance to the camera. It is important to mention that some areas of the mesh that are outside the frustum are also tessellated in order to avoid T-vertices, as we explained when describing our proposal. From a different perspective, in this case we have tested our method with a heightmap in geotiff format Sazid and Ramakrishnan (2003). Figure 4 shows the area that covers the map, which has a size greater than 1.5 GBytes and an error of around 25 meters. This terrain is located in Spain and has been extracted from a public web service. The whole map was initially allocated in main memory. In case of requiring more space than that available in main memory, it would be compulsory to resort to out of core techniques Silva et al. (2002); Varadhan and Manocha (2002). On the GPU side, graphics memory has a limited size (512 Mbytes in the graphics card that we have used). Thus, we also made use of an OpenGL extension (PBO or Pixel Buffer Object Elhassan (2005)) that enabled us to stream the heightmap from the CPU to the GPU, as commented in Section 3.2.

4.2 Performance

In order to evaluate the performance of our tessellation technique, we have conducted some tests where an initial mesh composed of 4 triangles is tessellated. The detail of the input mesh is first increased and later coarsened following a smooth trajectory of the camera.

Figure 8 presents the time needed for tessellating and rendering the input mesh at different tessellation levels. In this case the tessellation depends on the distance to the camera. Table 1 presents the results obtained in this test, helping us to show how the calculations for tessellation suppose an average increase of 60%.

For offering further tessellation experiments, Figure 9 presents the results of a similar test where all the geometry is tessellated at the same time, without any specific criterion. In this case, the obtained geometry will be composed of 2^n triangles, where n is the tessellation step. In this case, we can observe how the cost of the tessellation is exponential, offering very high temporal costs when outputting a large number of triangles. Again, the results are depicted in Table 2 to help us analyse the way this tessellation algorithm works. It is worth mentioning that, in our simulation, we will never include so many triangles as only those areas that need detail will be tessellated. Nevertheless, we considered it to be interesting in order to show how the temporal cost of the algorithm can be affected by the quantity of output triangles.

4.3 Coherence exploitation

An important contribution of the proposed approach is the possibility of exploiting coherence among the extracted tessellations. Table 3 presents the temporal results of a scenario similar to that presented in Figure 8, where the distance to the camera is used to guide the tessellation. These temporal costs include visualisation and tessellation of the input mesh. The column on the right offers the results without coherence maintenance, which nearly double the cost of our coherence-based algorithm. These results show that we can offer better performance as our tessellation scheme can exploit coherence among extracted tessellation, in contrast to previous solutions which had to start again from the input mesh.

5 Conclusions

In this article we have presented a new fully-GPU tessellation technique which offers view-dependent approximations. The scheme proposed avoided the appearance of T-vertices and other artifacts that can produce holes in the surface of a terrain. Another important aspect of this tessellation algorithm was the coherence exploitation, as it is capable of reusing the latest approximations when refining and coarsening the mesh. In this sense, we minimise the operations to perform in both cases, reducing the temporal cost involved in the tessellation process. This coherence maintenance is possible by storing some small pieces of information in each triangle, which is sufficient for altering the level of detail. It is important to underline that previous solutions were not capable of managing coherence, and thus entailed costlier tessellation processes. In addition, we have also considered a simple yet efficient approach to manage the heightmap information on the GPU.

A triangle-based criterion.....

For future work we would like to use larger terrains and consider out-of-core meshes, where all the geometry of the mesh does not fit within the memory on the GPU. From a different perspective, the appearance of Directx 11 involves further advances in computer graphics. Among the new stages of the rendering pipelines, we could highlight the tessellation unit, which will be able to produce semi-regular tessellations (Tariq, 2009) by itself. In this sense, for future work we would like to study the possibilities offered by the new tessellation units, in order to adapt our algorithm to this new framework.

Acknowledgements

This work has been supported by the Spanish Ministry of Science and Technology (projects TSI-2004-02940, TIN2007-68066-C04-02 and TIN2007-68066-C04-01) by Bancaja (project P1 1B2007-56) and by ITEA2 (project IP08009).

References

Apu, R. A., Gavrilova, M. L., 2004. Gtvis: Fast and efficient rendering system for real-time terrain visualization. In: International Conference on Computational Science and Applications (ICCSA). pp. 592–602.

Asirvatham, A., Hoppe, H., 2005. Terrain rendering using GPU-based geometry clipmaps. In: GPU Gems 2. pp. 27–45.

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.

Bosch, J., Goswami, P., Pajarola, R., 2009. Raster: Simple and efficient terrain redering on the GPU. In: Proceedings Eurographics Areas Papers. pp. 35–42.

Boubekeur, T., Schlick, C., 2005. Generic mesh refinement on GPU. In: HWWS '05: Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware. pp. 99–104.

Buatois, L., Caumon, G., Lvy, B., 2006. GPU accelerated isosurface extraction on tetrahedral grids. In: International Symposium on Visual Computing. pp. 383–392.

Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R., 2003. Planet-sized batched dynamic adaptive meshes (p-bdam). In: VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03). p. 20.

Clasen, M., Hege, H.-C., 2006. Terrain rendering using spherical clipmaps. In: EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization. pp. 91–98.

Cohen-Or, D., Levanoni, Y., 1996. Temporal continuity of levels of detail in delaunay triangulated terrain. In: VIS '96: Proceedings of the 8th conference on Visualization. pp. 37–42.

Dachsbacher, C., Stamminger, M., 2004. Rendering procedural terrain by geometry image warping. In: Proceedings of Eurographics Symposium on Rendering. pp. 103–110.

De Berg, M., Cheong, O., Van Kreveld, M., Overmars, M., 2008. Computational Geometry: Algorithms and Applications. Springer-Verlag.

Delaunay, B., 1934. Sur la sphre vide. a la memoire de georges voronoi. Otdelenie Matematicheskih i EstestvennyhNauk 7, 793–800.

Duchaineau, M., Wolinsky, M., Sigeti, D., Miller, M., Aldrich, C., Mineev-Weinstein, M., 1997. Roaming terrain: real-time optimally adapting meshes. In: VIS '97: Proceedings of the 8th conference on Visualization. pp. 81–88.

Elhassan, I., 2005. Fast texture downloads and readbacks using pixel buffer objects in opengl. http://developer.nvidia.com/system/files/akamai/gamedev/docs/Fast_Texture_Transfers.pdf?display=style-table&download=1.

Guthe, M., Balázs, A., Klein, R., 2005. GPU-based trimming and tessellation of nurbs and t-spline surfaces. ACM Transactions on Graphics 24 (3), 1016–1023.

Hoppe, H., 1998. Smooth view-dependent level-of-detail control and its application to terrain rendering. In: VIS '98: Proceedings of the conference on Visualization '98. pp. 35–42.

Johanson, C., 2004. Real time water rendering-introducing the projected grid concept. Tech. rep., Master of Science Thesis, Lund University.

Larsen, B. D., Christensen, N. J., 2003. Real-time terrain rendering using smooth hardware optimized level of detail. The Journal of WSCG 11 (2), 282–289.

Levenberg, J., 2002. Fast view-dependent level-of-detail rendering using cached geometry. In: VIS '02: Proceedings of the conference on Visualization '02. pp. 259–266.

Lindstrom, P., Cohen, J. D., 2010. On-the-fly decompression and rendering of multiresolution terrain. In: I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games. pp. 65–73.

Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., Turner, G. A., 1996. Real-time, continuous level of detail rendering of height fields. In: SIGGRAPH '96. pp. 109–118.

Liu, X., Rokne, J. G., Gavrilova, M. L., 2010. A novel terrain rendering algorithm based on quasi delaunay triangulation. Visual Computer 26, 697–706.

Livny, Y., Sokolovsky, N., Grinshpoun, T., El-Sana, J., 2008. A GPU persistent grid mapping for terrain rendering. The Visual Computer 24 (2), 139–153.

Losasso, F., Hoppe, H., 2004. Geometry clipmaps: terrain rendering using nested regular grids. ACM Trans. Graph. 23 (3), 769–776.

McConnell, J., 2006. Computer Graphics: Theory Into Practice. Jones and Bartlett Publishers.

Pajarola, R., 1998. Large scale terrain visualization using the restricted quadtree triangulation. In: VIS '98: Proceedings of the conference on Visualization '98. pp. 19–26.

Pajarola, R., Gobbetti, E., 2007. Survey of semi-regular multiresolution models for interactive terrain rendering. Vis. Comput. 23 (8), 583–605.

Pomeranz, A. A., 2000. Roam using surface triangle clusters (rustic). Tech. rep., University of California at Davis.

Rabinovich, B., Gotsman, C., 1997. Visualization of large terrains in resource-limited computing environments. In: Proceedings of the 8th conference on Visualization. pp. 95–102.

Rebollo, C., Remolar, I., Chover, M., Ramos, J. F., 2004. A comparison of multiresolution modelling in real-time terrain visualisation. In: ICCSA (2). pp. 703–712.

Sazid, S., Ramakrishnan, R., 2003. GeoTIFF - A standard image file format for GIS applications. http://www.geospatialworld.net/images/pdf/117.pdf.

Schmiade, T., 2008. Adaptive GPU-based terrain rendering. Master's thesis, Computer Graphics Group, University of Siegen.

Schneider, J., Boldte, T., Westermann, R., 2006. Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In: Vision, Modeling and

Visualization 2006. pp. 145–152.

Schneider, J., Westermann, R., 2006. GPU-friendly high quality terrain rendering. The Journal of WSCG 14 (1-3), 49–56.

Shiue, L., Jones, I., Peters, J., 2005. A real-time GPU subdivision kernel. ACM Transactions on Graphics 24 (3), 1010–1015.

Silva, C. T., Chiang, Y. J., El-Sana, J., Lindstrom., P., 2002. Out-of-core algorithms for scientific visualization and computer graphics. In: IEEE Visualization Conference 2002.

Szirmay-Kalos, L., Umenhoffer, T., 2008. Displacement mapping on the GPU - State of the Art. Computer Graphics Forum 27 (1).

Tariq, S., 2009. D3D11 tesselation. Game Developers Conference. Session: Advanced Visual Effects with Direct3D for PC, http://developer.download.nvidia.com/presentations/2009/GDC/GDC09_D3D11Tessellation.pdf.

Varadhan, G., Manocha, D., 2002. Out-of-core rendering of massive geometric environments. In: Proceedings of the conference on Visualization'02. pp. 69–76.

List of Figures

Number of triangles	Visualisation	Visualisation + Tessellation
4	1.45	2.28
16	1.46	2.29
64	1.48	2.28
256	1.58	2.45
1,024	1.71	2.75
3,644	1.83	3.16
5,756	1.88	3.95
3,644	1.83	3.39
1,024	1.71	2.75
256	1.58	2.45
64	1.48	2.28
16	1.46	2.28
4	1.45	2.28

Table 1
Comparison of time (in milliseconds) required for visualising and tessellating the input mesh using a distance criterion, by first increasing and then decreasing the detail following a smooth camera trajectory.

Number of triangles	Visualisation	Visualisation + Tessellation
4	1.45	2.29
16	1.46	2.29
64	1.48	2.44
256	1.58	2.44
1,024	1.71	2.76
4,096	1.80	3.17
16,384	2.08	4.56
65,536	2.71	6.42
262,144	4.89	9.31
562,500	7.05	15.23
262,144	4.89	10.96
65,536	2.71	6.83
16,384	2.08	5.16
4,096	1.80	3.59
1,024	1.71	2.76
256	1.58	2.8
64	1.48	2.44
16	1.46	2.29
4	1.45	2.29

Table 2
Comparison of time (in milliseconds) required for visualising and tessellating if completely tessellating the mesh, by firs increasing and then decreasing the detail following a smooth camera trajectory.

Number of Triangles	Coherence Exploitation	No Coherence Exploitation
16	2.29	2.49
64	2.29	3.63
256	2.28	4.89
1,024	2.75	5.46
3,644	3.16	7.55
5,756	3.95	8.04

Table 3
Performance comparison (visualisation and tessellation) with and without exploiting coherence (in milliseconds).

Figure2-eps-converted-to.pdf

Fig. 1. Tessellation patterns (Schmiade, 2008). The red colour indicates the edges that need refinement.

Figure1-eps-converted-to.pdf

Fig. 2. Example of *t-vertex* (v_5) after a tessellation step.

Figure3-eps-converted-to.pdf

Fig. 3. Tessellation example with the *id* value of each triangle. The $patterinInfo$ value of each triangle is also shown.

Fig. 4. Workflow to process and render a terrain surface in the GPU by using our approach.

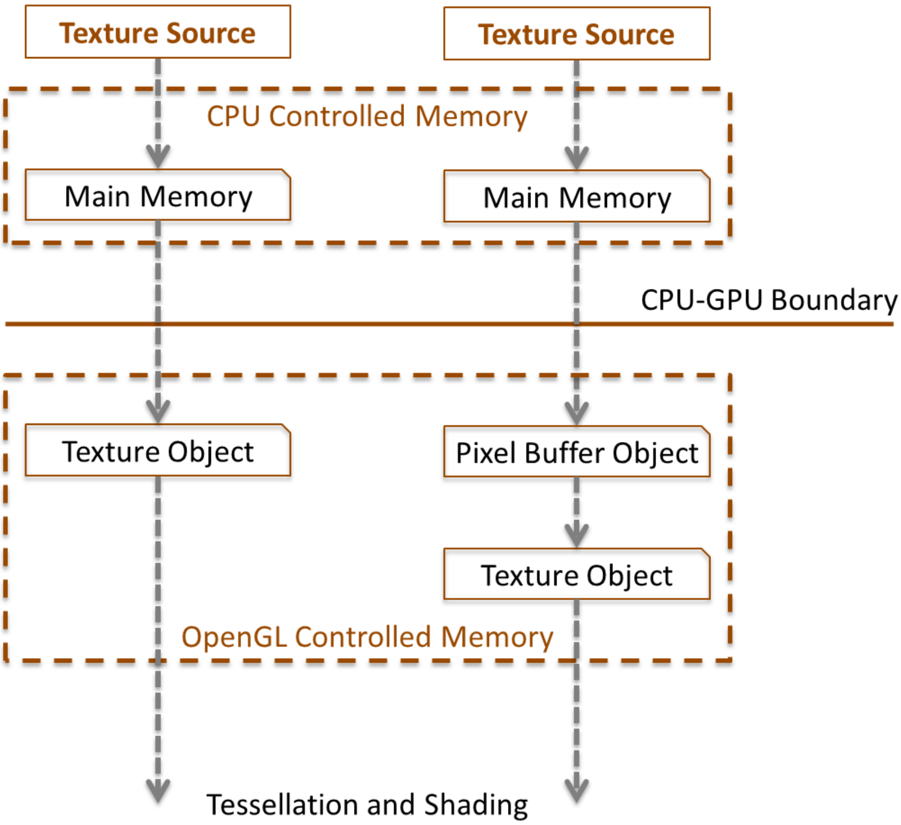

Fig. 5. Graphical comparison between the conventional manner to load a texture into the graphics memory and the alternative one by using Pixel Buffer Objects, which enable us to perform asynchronous operations with no CPU intervention.

Figure5-eps-converted-to.pdf

Fig. 6. Sample tessellation using a heightmap to modify the terrain surface.

Figure4-eps-converted-to.pdf

Fig. 7. Sample tessellation guided by a simulated frustum and using a heightmap from Spain obtained from a public web service. Geometry is refined up to 2,348 triangles.

Figure6-eps-converted-to.pdf

Fig. 8. Performance obtained using a distance criterion.

Figure7-eps-converted-to.pdf

Fig. 9. Performance obtained when completely tessellating the mesh.