LOD terrain rendering by local parallel processing on GPU

Alexandre Valdetaro[1] Gustavo Nunes[1] Alberto Raposo[1] Bruno Feijo[1] Rodrigo de Toledo[2]

[1]Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Dept. of Informatics, Brazil
[2]Federal University of Rio de Janeiro (UFRJ), Dept. of Computer Science, Brazil

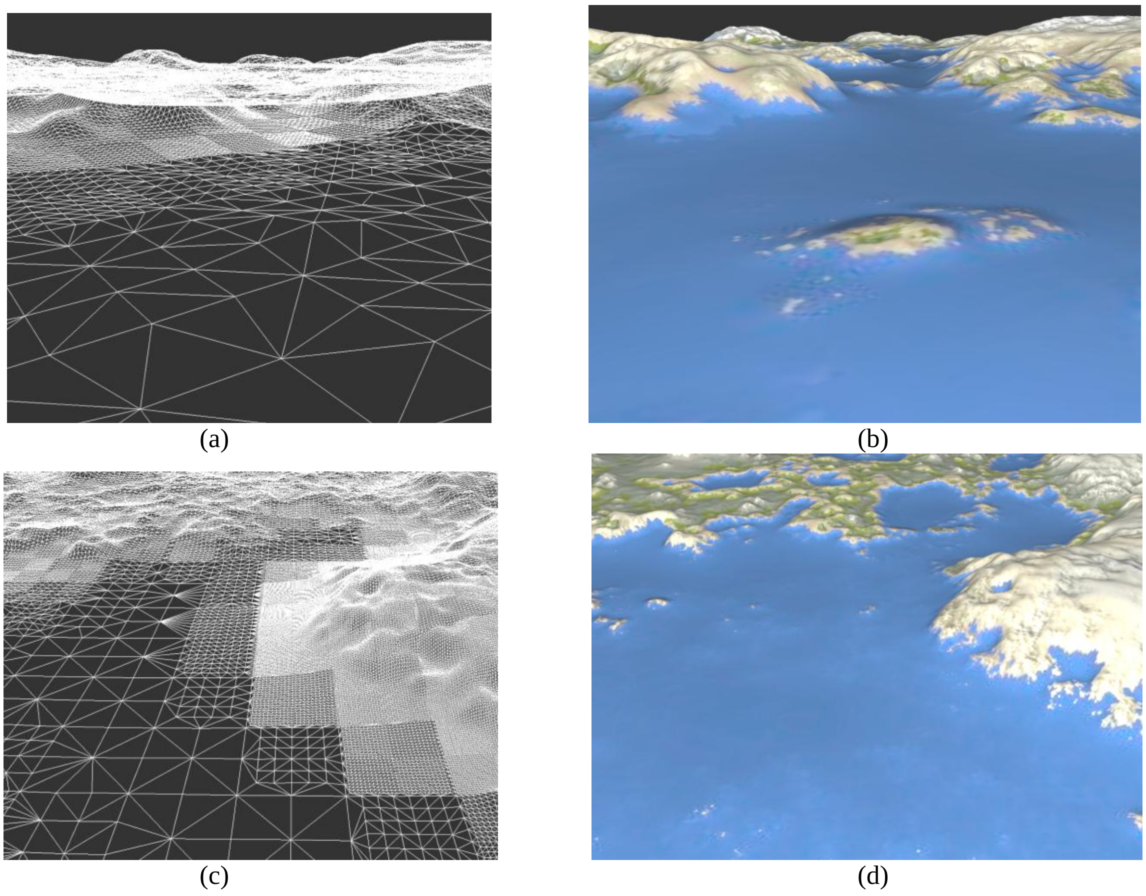

Figure 1: Terrain wireframe models and corresponding final visualization generated by the proposed technique running on an Nvidia GTX480 and an Intel i7 core. (a) and (b): 65536 x 65536 height map, area of 8 x 10^{12} m^2 with precision of 40*m* x 40*m*, at 52-109 *fps*. (c) and (d): height map 2048 x 2048, area of 4.4 x 10^{10} m^2 with precision of 100*m* x 100*m*, at 347-399 *fps*.

Abstract

In this paper, we present a new technique for highly efficient terrain rendering using continuous view-dependent Level-of-Detail based on hardware tessellation unit found in modern GPUs. Our technique is based on parallel local processing, in the sense that the results at each terrain patch do not depend on results already obtained at other patches. This patch-by-patch processing uses no hierarchical structure whatsoever, what makes it specifically tailored for GPU-based LOD terrain and is highly scalable.

Authors' contact:
avporto@tecgraf.puc-rio.br
gbnunes@tecgraf.puc-rio.br
abraposo@tecgraf.puc-rio.br
bfeijo@inf.puc-rio.br
rtoledo@dcc.ufrj.br

Keywords: LOD terrain, GPU

1. Introduction

Terrain rendering at interactive rates is essential for GIS applications, flight simulators, ground vehicle simulators, and games. However, despite the advances of the last decade, well-balanced control between high image quality and high processing time is still a challenge. A recent work (Asirvathan and Hoppe 2005) reports 90 fps for an interactive flight over a 20-billion-sample height map (216,000 x 93,600), but several restrictions are applied. Astonishing visual fidelity and automatic LOD are promised by Direct X11 tessellation support for the new generation of graphics cards (NVIDIA 2010), but the use of this facility is not straightforward, and innovative LOD techniques are yet required.

Terrains are a unique type of model, which is represented by a height map – a set of height samples over a planar domain. A naive approach for terrain rendering is to create a regular polygon grid and displace its vertices according to the height map, which is neither efficient nor scalable.

Algorithms for efficient terrain rendering have been proposed for over a decade, mostly involving hierarchical structures associated with LOD and culling techniques. Good surveys can be found elsewhere (De Floriani *et al*. 1996) (Pajarola and Gobbetti 2007).

Efficiency is required locally, as a percentage of the terrain may have constant height and does not need to be tessellated. However, the vertices should not be reduced naively, because there may be portions of the terrain that have a high frequency height variation – what demands a refined mesh for a reliable representation. The surface mesh must be automatically adapted to selectively refine or coarsen it according to the camera viewpoint. This requires a view-dependent LOD control that keeps consistent connectivity (*i.e.* no cracks) and smooth animation (*i.e.* no shimmering or popping artifacts). However, most real-time vertex splitting and edge collapsing (predominant in LOD techniques) require sequential CPU algorithms to update data-structures and, as a consequence, they are hard to be implemented efficiently, although there are some exceptions, such as (Losasso and Hoppe 2004). There is also the possibility of using the *Geometry Shader*, which has neighboring access and can update data structures residing in video memory. However, these approaches are very complex, because the mesh continuity needs to be preserved. Parallel view-dependent LOD control based on geometry shaders of modern GPUs has been recently proposed for arbitrary meshes (Hu *et al*. 2010). This control can be adapted to terrains, but the complexity of the solution is not significantly reduced.

Scalability is another critical issue when dealing with massive height maps. In this case, during close-up navigation, the viewer should receive the maximum possible refinement (*i.e.* one quad for every sample), which creates a critical overhead. Terrain rendering algorithms should be highly scalable. It has always been imperative to use some kind of hierarchical structure in order to navigate a terrain at highly variable scales. However, processes of building and accessing these data structures mean less computational efficiency and, sometimes, a burden to the CPU-GPU communication. Another type of scalability refers to the capacity that parallel algorithms should have to accommodate the increasing number of processing units made available in successive GPU generations.

In this paper, we present a new technique for highly efficient terrain rendering using continuous view-dependent Level-of-Detail based on hardware tessellation unit found in modern GPUs. Our technique is based on parallel local processing, in the sense that the results at each terrain patch do not depend on results already obtained at other patches. This patch-by-patch processing uses no hierarchical structure whatsoever, what makes it specifically tailored for GPU-based LOD terrain and is highly scalable.

This paper is organized as follows. Section 2 discusses related work. An overview of the proposed approach is presented in section 3. The characteristics of the proposed model are presented in sections 4 e 5. Section 6 presents some implementation details. The results are shown in section 7 followed by a conclusion section.

2. Related Work

Multiresolution techniques for terrain rendering can be classified into three basic classes:

(i) multiresolution models over irregular meshes (De Floriani *et al*. 1996), and
(ii) multiresolution models that exploit a certain semi-regularity of the data (Pajarola and Gobbetti 2007).
(iii) hybrid multiresolution models that use a regular structure for irregular meshes(Toledo et. al 2001).

Following a classification proposed by Pajarola and Gobbetti (2007), the second class of models can be grouped into 3 main lines:

[1] straightforward triangulations, such as tiled blocks (*e.g.* Falby *et al*. (1993)) and nested regular grids (*e.g.* Losasso and Hoppe (2004) and Asirvatham and Hoppe (2005)),
[2] triangulations based on trees (*e.g.* Lindstrom *et al*. (1996) and Duchaineau *et al*. (1997)). and
[3] cluster triangulations (*e.g.* Schneider and Westermann (2006), Cignoni *et al*. (2003), and Levenberg (2002)).

The first line uses regular grids, so they are scalable, simple to implement, and specially tailored for graphics hardware. The second line uses meshes with semi-regular connectivity, which relies on more powerful data structures. The third line works with regular or semi-regular contiguous portions of the mesh forming square or triangle patches. These patches are tessellated within the GPU with minimum CPU communication.

The model proposed in this paper uses a regular 2D grid of square patches (called *tiles*) that are independently tessellated by the GPU. This approach classifies the model as a cluster triangulation but with characteristics of simplicity found in straightforward triangulations. The key difference of our work from all the other techniques is that our system is the first to perform real-time LOD terrain rendering without using any kind of hierarchical data structure.

Non terrain-specific LOD strategies are also applicable to terrain cases. Many of those rely on the *Geometry Shader*, and can be very efficient. There are

GPU-based methods such as (Hu *et al*. 2010) (DeCoro and Tatarchuk 2007) and (Ji *et al*. 2006). Even though such strategies are GPU-based and very efficient, they rely on complex hierarchical structures and conditions for mesh updating to avoid mesh fracture.

Most of the new GPU-based proposals for multiresolution models avoid precomputation steps, but still require to traverse the entire representation at every frame (Ji *et al*. 2006) (DeCoro *et al*. 2007). A notable exception is the work by Hu *et al*. (2010), which, however, uses a hierarchical data structure – what makes it more complex than our system. To traverse the entire representation at every frame is not appropriate for dynamic change of the terrain – a critical issue in 3D games. In our model, the terrain can be deformed dynamically by direct manipulation of the height map with no extra burden to the system.

Our system performs parallel local processing based on hardware tessellation unit found in modern GPUs. Hardware tessellation is not something new. ATI and Xbox 360 game developers have had the opportunity to use it since 2005 at least. DirectX 11, recently launched in the market, brings tessellation support as a native feature. However, in 2007, ATI extended the Direct3D 9 API to provide access to the tessellation functionality via a wrapper API.

Despite all the previous advance in tessellation hardware, LOD terrain researchers seem not be aware of the problems game developers face to use this functionality. Indeed, current Xbox games have not being extensively reporting the use of this functionality. We think that the reasons for this lack of popularity are the problems to control shimmering and popping artifacts emerging from the straightforward use of tessellation hardware. The present paper proposes a model to overcome those difficulties through simple control and error criteria. We can find some similarity of our criteria with techniques proposed by Tatarchuk *et al*. (2010). However, our approach is broader than the one found in that reference, because we calculate a minimum and maximum subdivision for the edges and also for the center of every patch.

Our model establishes a clearer framework for the practice of LOD terrain rendering by local parallel processing.

To the best of our knowledge, no other current work in the literature provides a way to implement a robust and efficient LOD terrain rendering without hierarchical data structures. Also no other work establishes a clear framework for local parallel processing of terrain rendering.

3. Approach Overview

Refining the tiles view-dependently and continuously is a challenging problem, especially if the mesh tiles should not be refined the same way. We must guarantee mesh continuity. Edges of adjacent tiles must have equal number of subdivisions, regardless of each connected tile's number of sub-quads.

In order to refine the mesh on-the-fly, our approach makes use of the *Tessellator* pipeline stage introduced in recent GPUs (NVIDIA, 2010) (Tatarchuk *et al*. 2010), which is able to subdivide a tile (*i.e.* a square patch) in up to 4096 sub-tiles (also called "quads"). Tessellation and all its supporting stages are performed in parallel on the GPU, and the tessellation patterns are all performed by the fixed function part of the pipeline. Therefore, the idea is to tessellate the tile independently of the results of the other tiles, establishing a true local parallel processing.

We can achieve this independency by defining a regular grid in which an edge shared by two tiles has the same value of tessellation factor (called TessFactor).

We define tessellation reference points (Figure 2) and the tessellation factor for them is given by:

$$T_i = f(h_i, view, \nabla^2 h) \quad i = 1, 4 \qquad \text{Eq. 1}$$

where h_i is the height value corresponding to the point i, *view* is the camera position and $\nabla^2 h$ is the second-order derivative of the height map $h(u,v)$ corresponding to the tile. The location of a reference point on an edge corresponds to the middle of the edge. The fifth point is at the center of the tile, and its tessellation is given by:

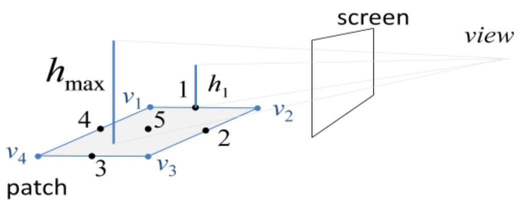

Figure 2. Tessellation reference points (four points in the middle of the edges and one at the patch center). vi is a patch vertex

$$T_5 = f(h_{max}, view, \nabla^2 h)$$

where h_{max} is the maximum height value of the samples inside the tile.

One of the principles behind Eq. 1 is that the relation between h_i and *view* should establish a minimum value of TessFactor (T_{min}) that guarantees an accurate tessellation from the camera's viewpoint. For example, if the camera is too far away from the patch and h_i is low, no tessellation will be necessary around the point i. Another principle underlying Eq. 1 is that $\nabla^2 h$ can be used to establish an upper bound for the TessFactor (T_{max}) to guarantee that patches will not be overly tessellated. For example, no tessellation is required for a flat terrain ($\nabla^2 h=0$). On the other hand, steep cliffs require high tessellation factors. According to these principles, Eq. 1 should calculate a TessFactor value in the interval $[T_{min},T_{max}]$.

The calculation of an edge TessFactor depends on the function $\nabla^2 h$ of the adjacent tile. However, this restriction does not destroy the local processing nature of the proposed method. Furthermore, we consider some simplifications in Eq. 1 that transform the calculation of the influence of $\nabla^2 h$ into a quite simple procedure.

4. Analysis the Height Map

4.1 Fractures Identification

A flat terrain would only need one primitive to be represented. However, real terrains have irregularities (such as steep cliffs) that need more geometry to be truthfully rendered. In this paper, we call these irregularities "fractures" (because of their "edge" look).

The process of finding fractures consists in averaging the derivatives of the height field in the horizontal and vertical direction (Figure 3). The calculation of the Laplacian $\nabla^2 h$ is not acceptable because the Laplacian is extremely sensitive to noise. Furthermore, we want a more simplified way of taking in account the steep variations of the height field. We propose a combination of two Sobel operators. The magnitude of the gradient vector calculated by the Sobel operator over the height map represents the height variation at a given pixel. However, the gradient is not enough to determine a fracture.

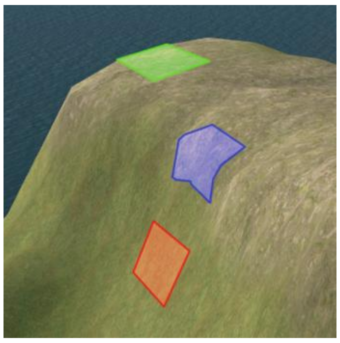

Figure 3. The top patch has low first and second derivative. The bottom patch has high first derivative and low second derivative. The middle patch has high second derivative. Second derivatives denote places that require refinement.

Places that should be viewed as fractures must have a change of height variation. In this paper, we estimate this rate by applying a Sobel operator over the gradient field obtained by the first Sobel operator. The new gradient values can be viewed as a "height acceleration map" (HAM, for short) that can be used to identify fractures. At the rendering phase, our algorithm needs to consult height accelerations in order to determine maximum refinement, which will be detailed in section 5.

4.2 Tile Size

The tile size is determined by the user, who decides how many samples from the height map one sub-tile (*i.e.* a quad) should cover. We have the most refined case when every quad covers one sample.

In real cases, a height map does not always require that every sample has one quad to be represented. However, if the density of texels is greater than the density of vertices of the tessellated patch, some undesirable patterns and shimmering artifacts may occur. These unwanted effects always occur when the sampling frequency (quad vertices) is too different from the signal frequency (texel density).

In our model we eliminate these problems by considering that 1 quad always covers 1 texel. In our system, this is guaranteed by a MIP mapping operation.

5. View-dependent LOD

5.1. Maximum Refinement: T_{max}

Our first concern is not to over-tessellate parts that do not require refinement. Intuitively, a flat terrain poses no need for further tessellation, so every tile that contains a nearly-flat part of the terrain should not be refined even when close-up. Flat terrain can be identified by the height acceleration map (HAM) presented in section 4.1. Also we need to identify fractures associated to the tile by looking over the HAM. In these two cases, we do not need an exact expression for Eq. 1, but only a reasonable proportionality between acceleration and TessFactor. We propose the following equation:

$$T_{max} = g(HAM_{max}) \qquad \text{Eq. 2}$$

where HAM_{max} is the maximum value of height acceleration associated over the tile and g can be a logarithmic function or the square root function. Figure 4 illustrates Eq. 2. In practice, HAM_{max} is within a range of values that do not contain the extremes of the domain interval.

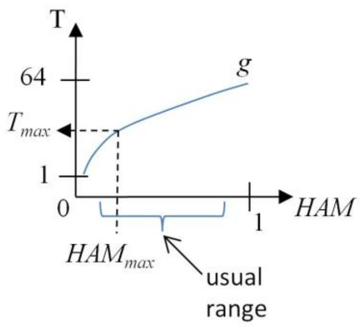

Figure 4. TessFactor as a function of *HAM*. *g* is not defined for T < 1, because the TessFactor range is [1,64]

Eq. 2 causes no burden on the complexity of the proposed algorithm. When the tessellation reference point is on a tile edge, the algorithm selects the biggest value of T_{max} between the two adjacent tiles.

5.2. Minimum Refinement: T_{min}

Another issue to be addressed is the minimum refinement required even from a long distance. A real example is a terrain part containing a peak. Every tile when viewed sideways from distance will be flat. Therefore, the terrain representation from long distances would have a whole tile as its deepest refinement, which is not a truthful representation. Thus, we need to estimate a minimum refinement per tile (T_{min}) based on the angle between camera and the tile's normal.

T_{min} is obtained from the control of the approximation error. It is common in the literature to measure the approximation error of a tessellated mesh by its deviation at the vertex. Duchaineau et al. (1997) and Lindstrom and Pascucci (2002) measured this deviation in the screen space. We also use the screen space, but in a different way. Figure 5 illustrates the proposed process, where we successively interpolate the point that should correspond to the maximum height h covered by the tile. We should notice that this maximum value (h) does not necessarily occur on one of the tessellation reference points (Figure 2). At each tessellation factor the point moves a little step towards the correct value. The deviation is monitored in the screen space and the process stops when the deviation is less than an allowed error $\hat{\varepsilon}$. If the process is a recursive one, the depth of the recursion is the value of T_{min}.

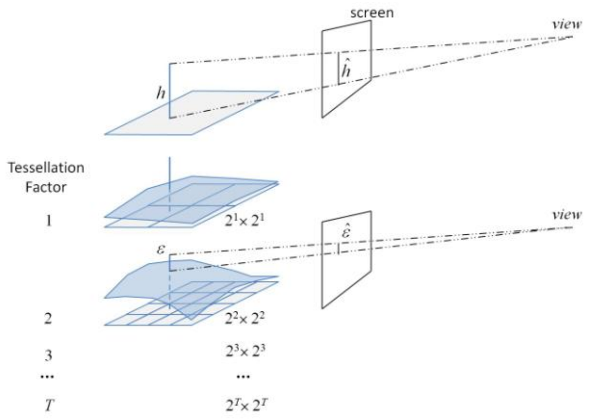

Figure 5 Process to calculate T_{min}, which stops when the projected error ε is equal to the allowed error $\hat{\varepsilon}$. h is the maximum heigh value.

However, the recursive process is not necessary, because a good estimate of T_{min} can be obtained by observing that $\hat{\varepsilon}/\hat{h}$ is proportional to 2^T . Without loss of generality, we can assume that

$$\frac{\hat{\varepsilon}}{\hat{h}} = 2^T \qquad \text{Eq.3}$$

and substitute Eq. 1 by the following equation:

$$T_{min} = log_2 \frac{\hat{\varepsilon}}{\hat{h}} \qquad \text{Eq.4}$$

5.3. Distance to Camera

Now, the value of the TessFactor for a tessellation reference point i is obtained from a calculation based on the camera distance d, keeping them inside the bounds of the maximum and minimum values. Here we assume an inverse proportionality between TessFactors and camera distances. Also we assume that there is a value of camera distance (d_{max}) that if $d < d_{max}$ then T should be equal to T_{max}. Considering these assumptions we propose the following equation to substitute Eq. 1:

$$T = \frac{T_{max} d_{max}^2}{d^2} \quad \text{if } d \in [d_{max}, d_{min}] \qquad \text{Eq. 5a}$$

$$T = T_{max} \quad \text{if } d < d_{max} \qquad \text{Eq. 5b}$$

$$d_{min} = \frac{\sqrt{T_{max}}}{\sqrt{T_{min}}} d_{max} \qquad \text{Eq. 5c}$$

Figure 6 illustrates Eq. 5.

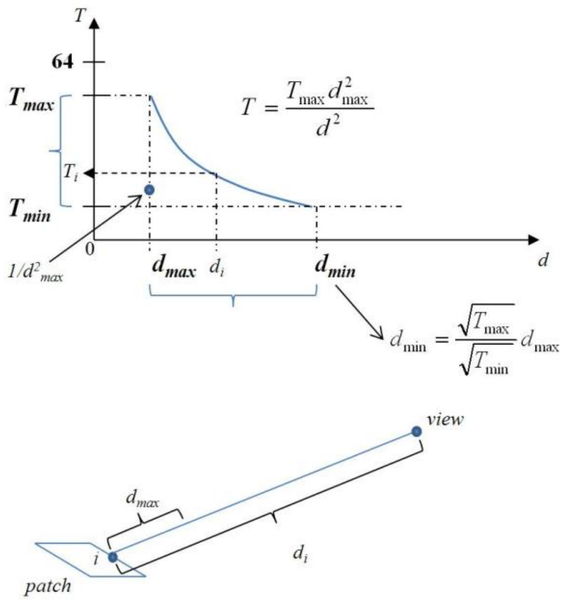
T
64
T_{max}
$T = \frac{T_{max} d_{max}^2}{d^2}$
T_i
T_{min}
0
$1/d^2_{max}$
d_{max}
d_i
d_{min}
d
$d_{min} = \frac{\sqrt{T_{max}}}{\sqrt{T_{min}}} d_{max}$
view
d_{max}
i
d_i
patch

Figure 6 TessFactor as a function of the camera distance d (Eq. 5)

6. Implementation

A coarse grid needs to be passed to the GPU so it can be tessellated later. The grid size is calculated based on the tile size as discussed in section 4.2. Besides position, we need also to pass the height map textures coordinates corresponding to each vertex of the quad.

6.1 Height acceleration map

Given a terrain represented by a height map H, our first goal is to find the rate of change of each *texel* in

H. In order to obtain such value, we use an approach based on the Sobel operator (Sobel and Feldman 1968). Two 3x3 kernels are convolved with *H* to calculate approximations of derivatives - one for horizontal and one for vertical changes. Assuming that D_x and D_y are the corresponding horizontal and vertical derivatives approximations, the calculation are as follows:

$$D_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * H \qquad \text{Eq. 6}$$

$$D_y = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * H$$

where * represents a convolution operation in two dimensions. At each point in the image, the gradient magnitude is obtained by combining the two approximations:

$$D = \sqrt{{D_x}^2 + {D_y}^2} \qquad \text{Eq. 7}$$

A new magnitude gradient map *M* is created. Each *texel* of *M* has the *D* value corresponding to the respective point at *H*. The same process is repeated with *M* to calculate approximations of the second derivative which we will call height acceleration map (HAM). The process to generate the HAM is similar to other image processing algorithms and it can be easily parallelized in the GPU using *pixel shaders* (Mitchell 2002).

6.2 Transferring the HAM to the GPU

The new GPU pipeline based on the *Shader Model 5.0* works with the concept of *patches*, which in our case is a tile of the terrain. The tessellation factors must be determined per *patch* at the *Hull Shader* stage.

Every patch needs four informative values (each one in a texture channel): HAM_{max} value, maximum height *h* , the texel position *x* of *h*, and the texel position *y* of *h*. Then a texture H with four data channels is created. The first channel receives the HAM_{max}. The second channel receives the maximum height value *h*. The third and forth channel receives the corresponding position (x,y) of the texel.

6.3 Terrain Displacement

After the mesh is tessellated, the following pipeline stage is the *Domain Shader*. The *Domain Shader* is invoked for each new vertex generated by the *Tessellator*. The Domain Shader receives the four initial patch vertices *v1,v2,v3,v4* and two normalized coordinates ([0..1]), *u* and *v*, which represent the generated vertices of that patch. To find the world position of the new vertex we linearly interpolate between the patch vertices, using *u* and *v* as interpolation factors. The same procedure is done with the vertex texture coordinates. After that, the vertical displacement of the vertex is sampled from the height map using the new vertex texture coordinates.

7. Results

Our tests were executed on an Intel Core i7 920 with an Nvidia GTX480 and 6GB of RAM. We started with a 2048x2048 height map, which is equivalent to the map of Rio de Janeiro ($4{,}4 \times 10^{10}\ m^2$) with a highest detail of $100\ m^2$. We used no frustum culling , which will not be the case in real applications.

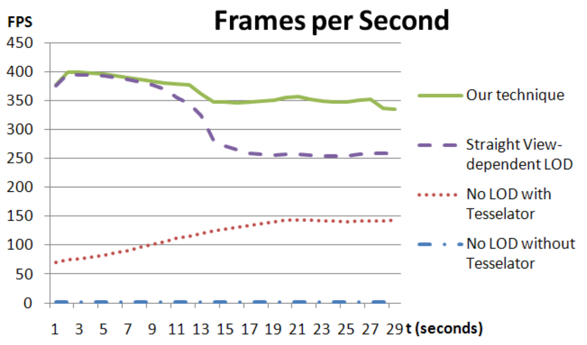

Figure 7 Frames per second (*fps*) for the case in Figures 1c and 1d. "Straight view-dependent LOD" does not use T_{max}, "No LOD with Tesselator" uses TessFactor 64 for the entire grid, "No LOD without Tesselator" is only a reference (the case where the entire mesh is transferred from CPU to GPU

Figures 7 and 8 shows the results for the 2048 x 2048 case (see Figures 1c and 1d in the first page). The results showed that just by being able to explore the Tesselator, without applying any LOD algorithm, our rendering is already much superior to a regular CPU based terrain algorithm in terms of fps, as can be seen in the two lower curves of Figure 7. In this case, both curves presented the same number of triangles (Figure 8). When comparing a straight view-dependent LOD with our technique, we are able to maintain consistent frame rates independent of the viewer position (Figure 7). Moreover, the triangle count is significantly reduced with our technique (Figure 8).

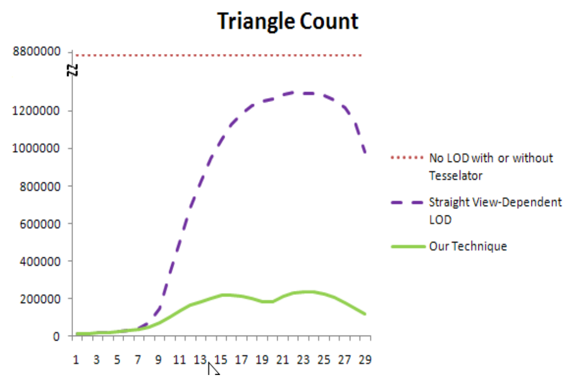

Figure 8. Triangle count for the same case of Figure 7.

Figure 9 presents fps values for the case of 65536x65536 height map, which is equivalent to the map of Brazil ($8\times 10^{12}\ m^2$) with a highest detail of $40\ m^2$. The result shows that our system is able to provide a consistent rendering of a massive terrain case. Such result indicates that our technique is applicable for the majority of the available height maps nowadays.

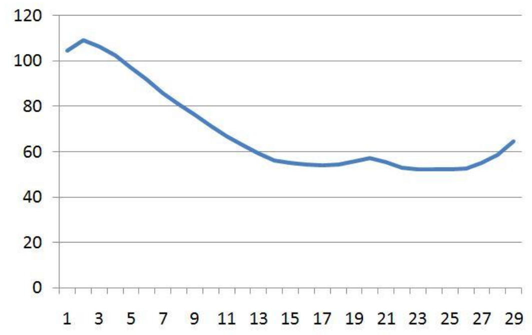

Figure 9. Frames per second (fps) for the 65536 x 65536 case in Figures 1a and 1b.

8. Conclusion and Future Work

In this paper, we present a new technique for massive terrain rendering using tessellation hardware, with no hierarchical structures. Our technique is capable of rendering massive height maps with interactive frame ratings and is very simple to implement. Also we present a true local parallel processing, in the sense that the results at each terrain patch do not depend on results already obtained at other patches.

With this approach we are able to determine, on-the-fly, the minimal tessellation needed for a minimal screen space error. Keeping the minimal possible refinement, we can reduce subdivision overheads. Consequently, on areas where deep refinement is required, we can use the tessellation hardware in its full potential. Making the best use of the tessellation hardware, we drastically minimize the CPU-GPU bus overhead with just a small set of primitives being transferred. Our algorithm is linearly scalable, while some other models present logarithmic hierarchical algorithms. However, our constant is very small due to the extensive usage of the hardware tessellation. In our tests, using the tessellation hardware we could render models in order of 10^8 triangles at interactive frame-rates with an error tolerance of 3 pixels. In any other older CPU Level-of-Detail approach this case would correspond to a prohibitive 5.0GB of data being transferred each frame between CPU and GPU. The continuous non-popping photorealism of the terrain is guaranteed by a smooth geomorphing and edge-splitting provided by the fixed tessellator stage of the GPU.

As a future work, we intend to create a hierarchy of terrain tiles that is able to keep the edges continuity. With such structure, the system would be able to select for every part of the terrain which level of the hierarchy to use. Thus the primitive count can be reduced drastically for massive terrains.

Acknowledgements

The authors would like to thank CNPq, CAPES, FAPERJ, Tecgraf, and ICAD/VisionLab for the financing support.

References

ASIRVATHAM, A. AND HOPPE, H., 2005. Terrain rendering using GPU-based geometry clipmaps. M. Pharr and R. Fernando (eds.), GPU Gems 2, Addison-Wesley, pp. 27-45.

CIGNONI, P., F. GANOVELLI, E. GOBBETTI, F. MARTON, F. PONCHIO AND R. SCOPIGNO, 2003. BDAM: batched dynamic adaptive meshes for high performance terrain visualization *In: Computer Graphics Forum, vol. 22, no. 3, pp. 505–514, 2003.*

DE FLORIANI, L., MARZANO, P., PUPPO, E., 1996. Multiresolution models for topographic surface description. *The Visual Computer*, 12(7), pp. 317–345.

DECORO, C. AND N. TATARCHUK, 2007. Real-time mesh simplification using the GPU. *In: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D '07), pp. 161–166, 2007*

DUCHAINEAU M., M.WOLINSKY, D. E. SIGETI, M. C. MILLER, C. ALDRICH, AND M. B. MINEEV-WEINSTEIN, 1997. ROAMing terrain: real-time optimally adapting meshes. *In: Proceedings of the IEEE Conference on Visualization (VIS '97), pp. 81–88, 1997*

FALBY, J., ZYDA, M., PRATT, D., MACKEY, L., 1993. NPSNET: Hierarchical data structures for realtime 3-dimensional visual simulation. *Computers & Graphics*, 17(1), pp. 65–69.

HU, L., SANDER, V.P., HOPPE, H., 2010. Parallel View-Dependent Level-of-Detail Control. IEEE Transactions on Visualization and Computer Graphics, Vol 16, No. 5, pp. 718-728.

JI, J., E. WU, S. LI, AND X. LIU, 2006. View-dependent refinement of multiresolution meshes using programmable graphics hardware. *In:* The Visual Computer, vol. 22, no. 6, pp. 424–433, 2006.

LEVENBERG, R., 2002. Fast view-dependent level-of-detail rendering using cached geometry *In: in Proceedings of the IEEE Conference on Visualization (VIS '02), pp. 259–266, 2002.*

LINDSTROM, P., D. KOLLER, , W. RIBARSKY AND HODGES, L.F., 1996. Real-Time, Continuous Level of Detail Rendering of Height Fields. *In: in Computer Graphics (Proceedings of SIGGRAPH 1996), pp. 109–118, 1996.*

LINDSTROM, P. AND V. PASCUCCI 2002. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *In: IEEE Trans. Visualization and Computer Graphics, vol. 8, no. 3, pp.239–254, 2002.*

LOSASSO, F. AND HOPPE, H., 2004. Geometry clipmaps: Terrain rendering using nested regular grids. In: *Proc. of ACM SIGGRAPH 2004*, ACM Trans. on Graphics, 23(3), pp. 769-776.

MITCHELL, J. L., 2002, Image processing with 1.4 pixel shaders in Direct 3D. In: *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*. Engel, W. F. (Ed.), Wordware, pp. 258-269.

NVIDIA, 2010. *NVIDIA GF100*. Available from: www.nvidia.com/object/IO_89569.html [Accessed 7 August 2010].

PAJAROLA, R. AND GOBBETTI, E., 2007. Survey on Semi-Regular Multiresolution Models for Interactive Terrain Rendering. *The Visual Computer*, 23(8), pp. 583–605.

SCHNEIDER, J., WESTERMANN, R., 2006: GPU-friendly high-quality terrain rendering. Journal of WSCG 14(1-3), pp. 49–56.

SOBEL, I. AND G. FELDMAN 1968. A 3x3 isotropic gradient operator for image processing. *In: Presentation for Stanford Artificial Project, 1968.*

TATARCHUK, N., BARCZAK, J. AND BILODEAU, B., 2010. Programming for real-time tessellation on GPU, AMD whitepaper. Available from: http://developer.amd.com/gpu_assets/Real-Time_Tessellation_on_GPU.pdf [Accessed 7 August 2010].

TOLEDO R., GATTASS M., VELHO L., 2001. Qlod: A data structure for interative terrain visualization. *Technical report, VISGRAF Laboratory, 2001. TR-2001-13.*