Error-Bounded Terrain Rendering Approach based on Geometry Clipmaps

Lucas Enrique Guaycochea and Horacio Antonio Abbate

Facultad de Ingeniería, Universidad de Buenos Aires
{lguaycochea,habbate}@fi.uba.ar

Abstract. This paper introduces a terrain rendering technique based on Geometry Clipmaps. This technique includes screen-space error analysis from application's view parameters in order to provide error-bounded visualization. This is accomplished dividing each nested patch into tiles to analyze the projected error into screen, as tiled-block terrain rendering techniques do. Finally, the implementation takes advantage of modern GPU processing power using DirectX 10 graphics library. The results obtained allow real-time navigation over large terrain extensions, consuming little CPU processing time.

1 Introduction

Applications, in which real-time rendering of virtual 3D environments is needed, always demand more realistic experience for its users. Among this sort of applications, we will consider those where the visualization of large terrain extensions is involved. From flight simulators to GIS applications, passing through any kind of outdoor game, techniques to process elevation data and generate real-time terrain rendering are needed to fulfill the requirements presented in those applications.

Terrains are modeled using a mesh of points to represent their surface. A naïve approach is to send the whole mesh to the graphics pipeline in order to render the terrain. Despite graphics pipelines' throughput has exponentially increased over recent years thanks to modern GPUs, the brute-force approach has strong limitations in the size of the terrains supported in order to achieve real-time rendering. Consequently, several techniques have been developed to support larger terrains where the mesh is assembled using regions with different levels of detail (LOD) in order to reduce the geometry introduced into the pipeline in each frame.

Approaches using LOD are possible since the fact that the perception of details of an object (e.g. a particular region of a terrain) decreases as the distance from the viewer to the object increases. The size of the projection of these details into the final image on screen, as a consequence of the view-perspective projection transformation, may be less than the pixel resolution of the output display. The election of the appropriate level-of-detail for each region in the terrain is taken from different motivations, as can be seen along the bibliography on the theme.

This work is based on a technique called Geometry Clipmaps. It was introduced by Lossaso & Hoppe [1] and then extended to a GPU-based implementation by Asirvatham & Hoppe [2]. The authors' proposal is to represent the terrain using a set

of nested regular grids of different LOD centered about the viewer. The nested grids have successive power-of-two resolutions and are translated as the viewer moves. The translation of the grids involves an incremental update of the elevation data of each nested grid.

Geometry Clipmaps decides the LOD over the terrain using only the 2D distance to the viewer. This strategy allows the technique to be independent from terrain local roughness and, therefore, to maintain the CPU work to the minimum and to guarantee constant throughputs (frame rate). Nevertheless, this approach has some limitations, as it is mentioned by its authors, to prevent the perception of changes in the surface of particular rough terrains as the viewer moves.

The technique proposed in this work is targeted to resolve the terrain rendering problem for applications where an immersive virtual reality on a well-known real-world environment must be provided to the user, such as flight simulators. In other words, the users must have an accurate real-world terrain perception without noticing any artifacts. In order to achieve this requirement, error-less or, at least, error-bounded surface terrain representation must be guaranteed by the solution.

The approach presented adds, to the state-of-the-art geometry clipmaps technique, the ability to analyze the error incurred in the use of a particular LOD in a region. The error is calculated projecting into screen-space the world-space error between the full-resolution region and the same region represented with lower detail. This projected error is calculated from the distance to the viewer and, also, from view parameters, such as the size of the window and the field-of-view of the camera. Then, the resulting screen-space error is compared with a pixel threshold defined by the application. This is done dividing each nested grid into tiles and then deciding if any of them needs to be refined to guarantee the error-bounded terrain representation. This strategy is taken from well-known tiled-blocks terrain rendering techniques [3, 4, 5, 6].

The error analysis discussed before, compared with the pure Geometry Clipmaps technique, involves more CPU process to compute the projected error and, if refinement of tiles is necessary, some CPU to GPU transfer of elevation data and more geometry to render. Even though, this obviously means a lower throughput rate from our technique, in modern hardware it performs with more than acceptable rates (see Section 3).

2 The terrain rendering technique

2.1 Terrain Representation

As it was introduced, the solution presented is based on the Geometry Clipmaps technique [1]. Then, the terrain is represented using a set of nested grids, that we call ***patches***, around the viewer's position. Each patch represents a region of the terrain with different resolution or level of detail. The most detailed level is zero ($L = 0$), where the spacing between the points of elevation data is at the highest resolution. Each following level covers four times more surface than the previous one, which

means that the former resolution doubles the latter. So, a patch resolution (g_L) is 2^L times the finest dataset surface resolution, for levels L = 0, 1, 2….

Moreover, all patches have the same amount of samples or vertices, *n x n*. In contrast with Lossaso and Hoppe [1], we use $n = 2^k + 1$ (where k = 1, 2, 3…), that is needed to divide each patch in tiles. The election of this value for *n* makes us handle nine cases of relative positions between successive patches. A patch center position must lie in a vertex that belongs to next coarser-level patch. In this way, patches edges can share vertices available in both, so as not to present discontinuities in the terrain.

In order to optimize the performance, we add to patches' render size *n*, some extra elevation data that is loaded to use as a border cache. We choose to have a power-of-two border size in each direction (top, bottom, left and right). This can be also useful if the elevation data is obtained from compressed resources that apply block-compression schemes.

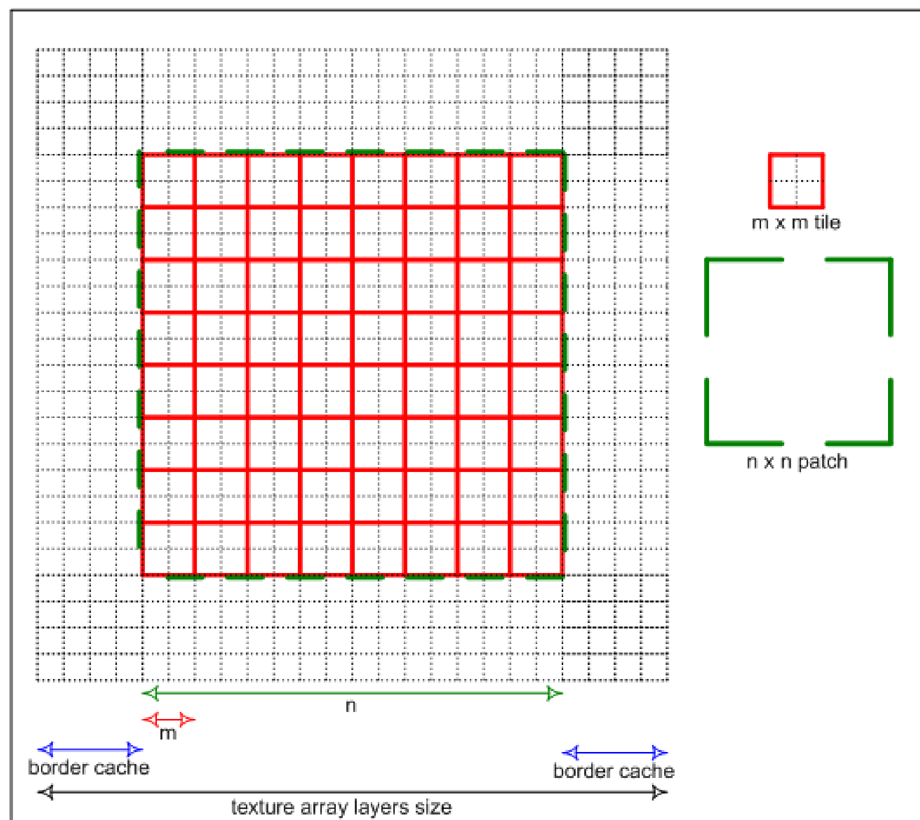

Fig. 1 Sizes of the different grids used for patches, tiles and layers of the texture array. Example: m = 3, n = 17.

Elevation data will be loaded to the GPU into texture arrays (available in GPUs that support Shader Model 4.0 [7] or later). Each patch has its elevation data loaded in a different layer of the texture array. Then, using vertex buffers describing 2D

“footprints” [2], the vertex’ z-value is sampled in the vertex shader from the corresponding layer in the texture array.

Recalling a key-point from geometry clipmaps [1, 2], the elevation data is incrementally updated using a toroidal access with 2D wraparound addressing. With this strategy, GPU-CPU bandwidth utilization is optimized as only new regions of elevation data is updated as the viewer moves. Moreover, the use of a border cache avoids frame-to-frame updates of few, or singles, rows or columns to the texture layers.

Up to this point, the base of the surface representation approach has been described, but it lacks of view-dependent and error-bounded screen-space error properties. In order to add these properties, we decided to divide each terrain patch into tiles. These tiles allow the error analysis using view-dependent based metrics (see section 2.2).

Square tiles of size *m x m* are used, where $m = 2^j + 1$ (with j = 1, 2, 3…). The solution needs to have the patch divided in, at least, 8 x 8 tiles. This means that $k - j \geq 3$. This is necessary to manage the center position of each patch, that must lie in a corner of an own tile. On other hand, a layer in the texture array used to render each patch must have an integer count of tiles, so the border cache size must be $2i(m-1)$ (with i = 1, 2, 3…).

Finally, Figure 1 shows relationships between the sizes of the different grids discussed in this section.

2.2 Screen-space error analysis

When the available terrain surface is approximated using a lower resolution mesh, some approximation errors will occur. The approximation error is defined as the vertical distance of a vertex present in the full-resolution mesh with respect to its interpolated position when it is removed in a particular level of detail with lower resolution. This approximation error that appears when the terrain is not represented using a full-resolution mesh is called the world-space error, as it is calculated from world-space coordinates.

World-space error can be measured in, both, relative and absolute terms. Some works, like Lindstrom et al. [8], measure it relatively between successive levels of detail. In order to satisfy accurately error-bounded property this measure must be correctly saturated [9]. Nevertheless, it is more accurate to calculate the absolute error against the full-resolution terrain, as it is done in ROAM [10].

In this solution, the world-space error is computed in a pre-process and saved into a small file. That file is loaded in the loading phase into CPU memory so as to avoid disk-access latencies. Our technique calculates absolute world-space error. The approach consists in dividing the terrain into *m x m* tiles at the different levels of detail and then saving the maximum world-space error found in each tile. Finally, as the viewer moves and the patches are incrementally updated, those pre-calculated maximized errors are queried.

The maximum world-space error for each tile, in a particular level of detail, is a necessary input to analyze if the approximation errors are perceptible to the viewer. Then, maximum screen-space error is conservatively calculated. From the distance

from the viewer to the closest point in the tile's bounding volume, and view parameters, as the viewport size and the field-of-view, world-space error is projected (using a perspective projection) onto the screen space viewing plane to obtain the screen-space error measured in pixels.

Finally, the application chooses a value for a tolerable screen-space error. This is used as a threshold value to compare with. If the projection of the tile's maximum world-space error onto screen-space exceeds this threshold, then that region of terrain needs to be represented with a higher resolution mesh (Figure 2).

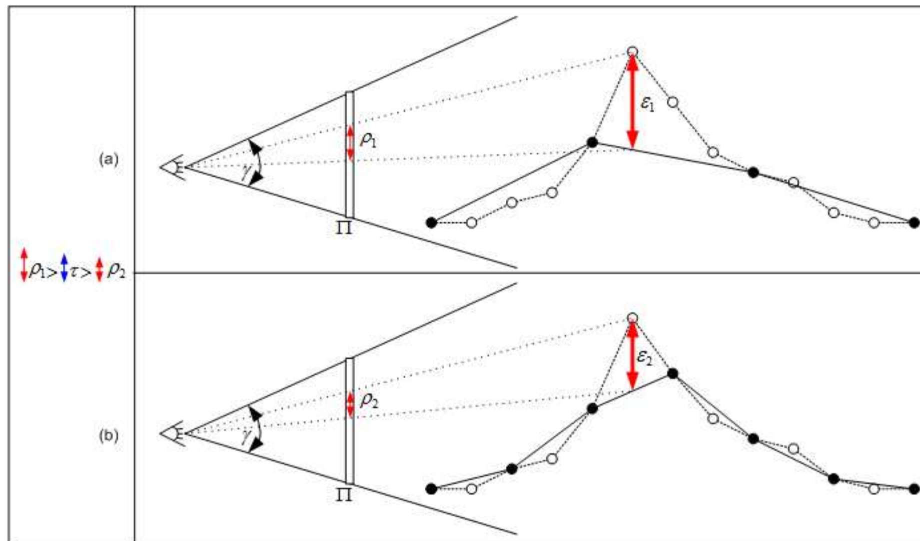

Fig. 2. World-space errors ε_1 and ε_2, originated from two different LOD representations, are projected into the projection plane Π. (a) Level 2 representation: maximum screen-space error ρ_1 exceeds threshold τ. (b) Level 1 representation: maximum screen-space error ρ_2 is smaller than threshold τ.

2.3 Rendering Strategy

In this section we will describe the high-level algorithm used to generate each frame in runtime. Some useful details on implementation will be given to provide a higher performance.

First, given the viewer's position, we calculate each patch center position and its elevation data is updated if necessary. Then, the tiles that cover the render surface (*n x n*) of each patch are tested against the view frustum. At least, a coarse view-frustum culling is absolutely necessary, since it decreases nearly to a quarter the geometric rendering load to the graphics pipeline (calculated for a field-of-view of 90 degrees). The test is done using an axis-align bounding box for each tile.

Then, the screen-space error analysis is performed on those tiles which were not discarded in the view frustum culling test. As explained, the maximum world-space error present in a tile, at a particular level of detail, is projected into screen space and compared with the threshold value. If the projection exceeds the threshold, that tile

needs to be represented with a higher resolution. The tile will be represented using a level of detail which maximum world-space error projection will result smaller than the threshold. That tile, that we call a ***refined tile***, will be rendered using a two, four, height, etc. times higher resolution mesh as needed (from experience it rarely needs more than two levels of refinement). Refined tiles have a size of $(r+1)*(m-1)+1$ *x* $(r+1)*(m-1)+1$, where *r* is the refinement level (*r = 1, 2, 3…)*.

The elevation data needed for the refined tiles is loaded into GPU memory. There is a texture array for each refinement level. Elevation data is loaded into the different layers which are managed using the least-recently-used memory management policy.

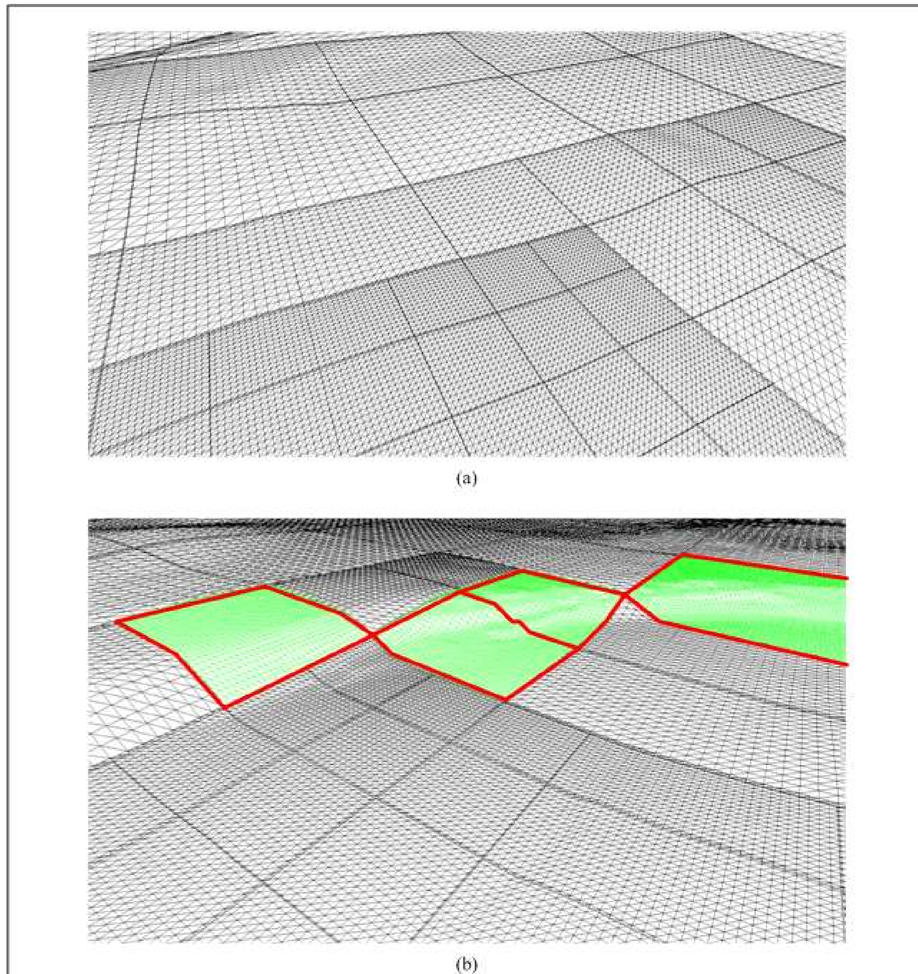

Fig. 3. Frames rendered by our solution in wire-frame mode (n = 129, m = 17). (a) Nested patches divided in tiles and different successive patches' relative position can be seen. (b) Refined tiles are drawn in green and marked in red.

Rendering is done taking advantage of the instancing technique available in modern GPUs [7]. Instancing is used with a dynamic vertex buffer filled with data to

instantiate each tile. It allows rendering several instances of a tile using a single *Draw* call, linking two vertex buffers to the pipeline input stage: the first containing the 2D footprint for each tile, and the second filled with the instances' data.

Finally, we render the tiles which do not need refinement first, then the refined tiles, grouped by refinement level. Figure 3 shows two frames rendered by our solution: in (a) the nested patches divided into tiles can be seen, and in (b) refined tiles are drawn in green and marked in red.

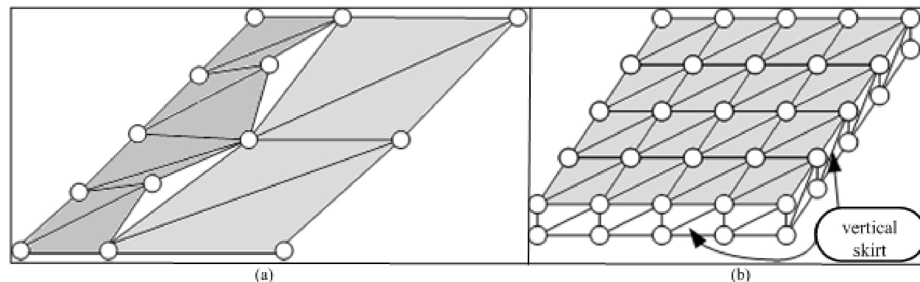

Fig. 4. (a) Gaps that may appear at the edge of tiles represented with different LOD. (b) Vertical skirts that are added around each tile.

2.4 Level-of-detail approaches problems

Some artifacts may be perceptible in terrain rendering when level-of-detail approaches are used.

First, cracks in the terrain surface may appear at the edges of two regions represented using different resolutions. This problem is originated in non-continuous LOD approaches as the one described in this paper; continuous LOD approaches as [8] and [10] do not present this problem.

Tiled-blocks techniques, and also Geometry Clipmaps, have to solve this problem. Tiles at different resolution which share an edge do not have the same quantity of vertex at the edges. This can lead to gaps in the edge of two neighboring tiles (see Figure 4 (a)). Some approaches [3, 5, 6] solve it modifying the connections of the vertices in one of the tiles (usually the one with higher resolution). We will not use this approach since it requires for each tile the knowledge of the resolution of its neighbors, adding complexity and the need to manage several cases. On other hand, Ulrich [4] describes some techniques to solve the problem adding geometry. The options are to add around each tile *"flanges"*, *"ribbons"* or *"skirts"*. Geometry Clipmaps [1, 2] uses zero-area triangles to cover the perimeter of each patch. However, this requires disabling back-face culling for terrain rendering. Finally, as proposed by Ulrich [4], we decided to use vertical skirts around each tile to prevent gaps (see Figure 4 (b)).

Another artifact that may occur in level-of-detail terrain representation is known as *"popping"*. It refers to the perception of a *"pop"* in the terrain surface, a change in the terrain geometry that suddenly happens. It occurs when there is a change in the level of detail used to represent a particular terrain region while the viewer is moving. To prevent this artifact from being noticeable, some approaches manage to *slowly* change the resolution. This means that the strategy is to interpolate the vertex height

between successive levels of detail when a change in the representation will occur. So, terrain geometry will slowly morph to (or from) a higher resolution representation, and for that reason, this is known as *geomorphing,* introduced by Hoppe [11] and used by many others.

As our technique has the error-bounded property, since we do not want to notice differences from the highest resolution terrain, choosing a low value for the screen-space error threshold will also guarantee that no popping will be noticeable. Pops can not exceed in screen the threshold chosen by the application.

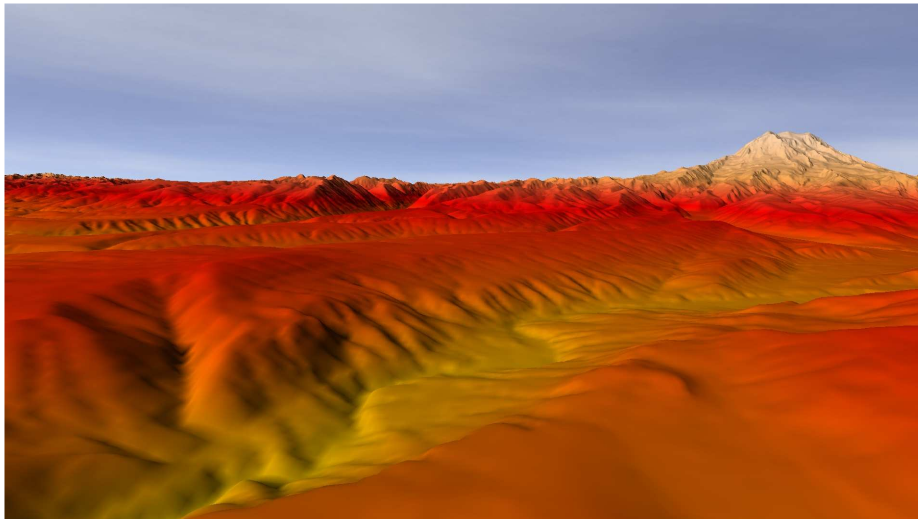

Fig.5: Rendering result on a fly through Puget Sound 16K x 16K dataset.

3 Results

Implementation was done using DirectX 10 graphics library [12] in order to access functionalities of GPU's Shader Model 4.0 [7]. Experimentation was performed using the well-known dataset of Puget Sound area. A 16,385 x 16,385 grid with 10 meters spacing covers a 163.85km x 163.85km terrain area which is suitable enough to applications, such as a flight-simulator. Height values have a 16-bit representation with 0.1m vertical resolution.

Table 1. Results from the five experiments are collected in this table.

	Run 1	Run 2	Run 3	Run 4	Run 5
Flying Time	132 sec	189 sec.	180 sec.	166 sec.	140 sec.
Frames Total Count	10960	15545	13564	13118	12575
Max. Frame Time	49.5 msec.	44.1 msec.	40.7 msec.	41.5 msec.	36.0 msec.
Frame Time in 5-10 ms	49.91%	43.94%	37.14%	41.31%	49.46%
Frame Time in 10-15 ms	24.80%	14.91%	31.21%	27.19%	29.11%

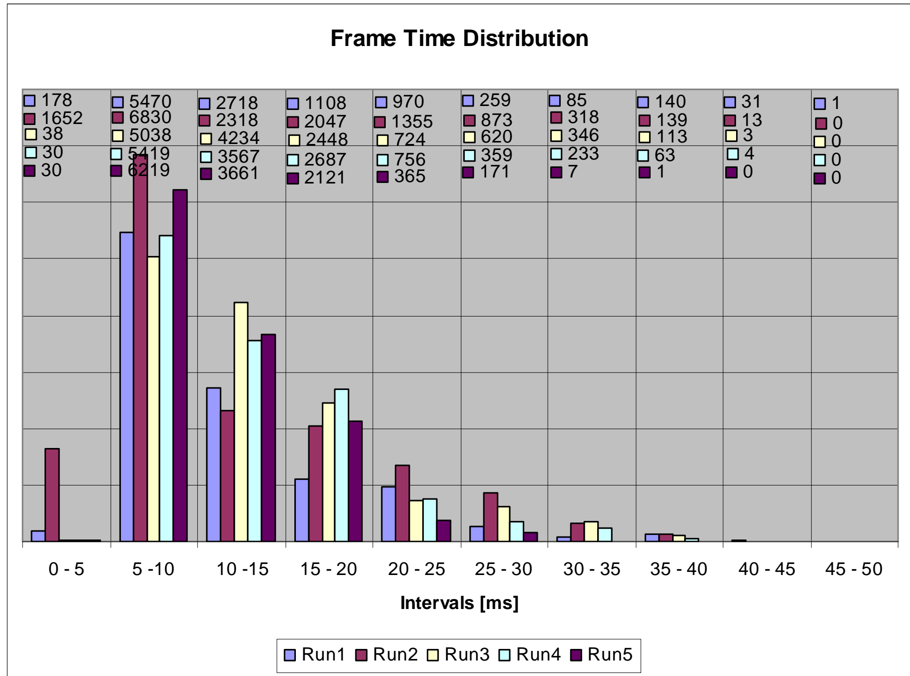

Fig. 6. Distribution of time used to generate each frame in five different flies over the terrain.

Results (Figure 5) were obtained using a PC, running Windows 7 OS, with a 2.8 GHz Intel® Core™2 Quad, 4GB system memory and an Nvidia GeForce GTX 280 graphics card with 2GB of video memory.

Application was configured to run in a 1920 x 1080 full-screen window and a viewer's horizontal field-of-view of 90 degrees. Choosing a threshold value of 5 pixels (0.5% of vertical resolution), we obtain an average of 120 frames/second, which means an average of 8 milliseconds to generate each frame. On other hand, when the viewer flies near high detailed regions, for example the mountain present near the center of Puget Sound terrain, the average frame rate drops to 50 fps. Due to refinement data loading, some frame time can reach about 40 milliseconds.

We run five experiments flying over the Puget Sound terrain at 340 meters per second, from the center of the terrain to the high-detailed mountain. We measured the time to generate each frame during a flying time between two and three minutes approximately. From Figure 6, we obtained that the 70% of the frame times are between 5 and 15 milliseconds (44.35% in 5-10 ms interval, and 25.44% in 10-15ms interval). At last, from Table 1, the maximum frame time was 49.5 milliseconds.

Finally, note that average frame time allows including this technique into a graphics engine, leaving free time to other processing and rendering tasks within each frame.

4 Conclusion

This paper introduces a terrain rendering approach that is based on the start-of-the-art terrain rendering technique called Geometry Clipmaps. The contribution of this solution is to introduce an approach where the limitation presented in Geometry Clipmaps, to represent particular rough terrain without noticeable surface changes, is resolved. The strategy consists in adding view-dependent screen-space error analysis to ensure screen-space error-bounded terrain representations.

The solution presented shows a good performance in modern hardware. As it consumes little CPU processing time it can be integrated into a graphics engine to resolve the terrain rendering problem.

Finally, the approach presented targets applications where an immersive environment needs to be represented over a user well-known real-world terrain surface, such as a flight-simulator.

References

[1] Lossaso, F., Hoppe, H.: Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. ACM Transactions on Graphics (SIGGRAPH) 23(3), 769-776 (2004).

[2] Asirvatham, A., Hoppe, H.: Terrain Rendering Using GPU-Based Geometry Clipmaps. GPU Gems 2, Chapter 2, Addison-Wesley, March 2005.

[3] de Boer, W.: Fast Terrain Rendering Using Geometrical MipMapping. E-mersion Project, October 2000.

[4] Ulrich, T.: Rendering massive terrains using chunked level of detail. In: Super-size-it! Scaling up to Massive Virtual Worlds (ACM SIGGRAPH Tutorial Notes). ACM SIGGRAPH (2000)

[5] Snook, G.: Simplified Terrain Using Interlocking Tiles. Game Programming Gems 2, pp. 377-383, Charles River Media, 2001.

[6] Wagner, D.: Terrain Geomorphing in the Vertex Shader. Shader X2, Wordware Publishing, 2003.

[7] Patidar, S., Bhattacharjee, S., Singh, J., Narayanan, P.: Exploiting the Shader Model 4.0 Architecture. *Technical Report IIIT Hyderabad*, 2006.

[8] Lindstrom, P, Koller, D., Ribarsky, W., Hodges, L., Faust, N., Turner, G.: Real-Time, Continuous Level of Detail Rendering of Height Fields. *Proceedings of SIGGRAPH 96*, 109-118. August, 1996.

[9] Pajarola, R., Gobbetti, E.: Survey on Semi-Regular Multiresolution Models for Interactive Terrain Rendering. The Visual Computer 23(8), 583-605, 2007.

[10] Duchaineau, M., Wolinsky, M., Sigeti, D., Miller, M., Aldrich, C., Mineev-Weinstein, M.: ROAMing Terrain: Real-time Optimally Adapting Meshes. *IEEE Visualization '97*, 81-88. November, 1997.

[11] Hoppe, H. Smooth view-dependent level-of-detail control and its application to terrain rendering. IEEE Visualization 1998, 35-42, October 1998.

[12] Blythe, D. Direct3D 10. GPU Shading and Rendering, SIGGRAPH 06 Course, 2006.