A Multi-threaded Multi-context GPU Methodology for Real-time Terrain Rendering

Mohammad A. Yusuf
Instinct Games
http://www.instinctgames.com
29 Beirut Street
Beirut Square
Heliopolis
11757, Cairo, Egypt
mhdyousif@gmx.net

Mostafa G. M. Mostafa
Department of Computer Science
Faculty of Computer and Information Sciences
Ain Shams University
11566, Cairo, Egypt
mgmostafa@cis.asu.edu.eg

Taha I. Elarif
Department of Computer Science
Faculty of Computer and Information Sciences
Ain Shams University
11566, Cairo, Egypt
taha_elarif@yahoo.com

ABSTRACT

Real-time rendering of large terrains has several important applications. Hence, many methods have been devised to solve this problem. The main challenge for such methods is to deal with a large terrain dataset and maintain interactive frame rates. In this paper, we propose a level-of-detail (LOD) based multi-threaded multi-context method that works on two separate activities. Each activity is assigned to its own CPU thread and GPU context. The LOD hierarchy is constructed on the GPU context of the errors activity and stored as a 2D texture map. This texture map is used by the blocks rendering activity via its CPU thread to initiate the rendering process by sending different terrain blocks as translation and scaling parameters to its GPU context, which uses a reusable single shared vertex and index buffer to render the required block based on the passed parameters and the height-field texture. The results show that the proposed method achieves high interactive frame rates at guaranteed very small screen-space errors.

Keywords

Real-time rendering, large terrain rendering, level of detail, multi-context programming, GPU programming

1 INTRODUCTION

Terrain rendering plays an important role in computer graphics applications such as synthetic vision systems (SVS), game engines, 3D geographic information systems, military simulation, battle command, flight simulation and surveying. Modeling complex outdoor environments has ever been a challenge using available computing hardware. The main problem has always been, without losing visual details, rendering terrain models that can be quite large requiring significant amount of processing power and storage.

This is problematic because the renderer must process a huge number of geometry representing the terrain every frame to maintain interactivity. Visualizing an enormous detailed terrain using brute force conventional rendering methods where everything is rendered first and then the screen is clipped to the visible portion is not currently possible at interactive frame rates.

Different methods have been devised to tackle this issue. Such methods can be classified into ray-tracing and triangulation methods. Ray-tracing methods are image order algorithms that depend on casting a ray for each pixel from the viewpoint through the terrain and calculating the final pixel color based on its intersection with the terrain without actually generating terrain geometry and as such they are not suitable for applications which use terrain geometry for other purposes like game engines and SVS. Triangulation methods are object order algorithms that depend on generating polygons from the terrain description and simplifying them before rendering using regular polygon rendering techniques. They often use some type of multilevel hierarchical structure to render the geometry in different levels of detail (LODs) based on a maximum level approximation error.

In this paper, we propose a multi-threaded quad-tree based triangulation method for rendering terrain completely on the GPU. All triangulation and approximation errors calculation tasks are done completely on the GPU freeing the CPU for other application-specific work. The terrain is divided into blocks of equal size and triangulation is done on-the-fly by the GPU using

a single shared vertex/index buffer for every block by varying its scaling and translation values based on its location and level of detail cutting down GPU memory costs to only one block of terrain.

This paper is organized as follows: Section 2 summarizes related work, the proposed methodology is detailed in Section 3, results are given in section 4, and finally we conclude the paper in section 5.

2 RELATED WORK

Terrain data is commonly represented as either a Triangulated Irregular Network (TIN) which is essentially a 3D data model or a height-field. A height-field is regularly spaced grid of elevation values. Common height-field based methods use ray-tracing or triangulation techniques.

Ray-casting techniques cast a ray for each pixel of the output image to determine its color independently. In [Dun79a], a ray is cast from the viewpoint through the pixel and the 2D projection of the ray across the base plane of the terrain grid is traced using line algorithms. At each tracing step, the height of the ray is compared to the height of the terrain data. When the ray height drops below the terrain, the intersection point is found and the input texture map is interpolated and sampled at the intersection point to give the final pixel color. Cohen-Or et al. in [Coh96b] optimized this further by using the found intersection point of the previous ray as an approximation of the starting point forthe new ray exploiting the fact that adjacent pixels have a good chance of being also adjacent in the terrain. These methods are iterative in nature because each ray trace must go through the whole process.

A run-based ray-traversal method is proposed in [Hen04a] that accelerates this process but it doesn't use spatial partitioning as done in [Coh93a] which used a quad-tree as a hierarchical data structure to store terrain data at different LODs where each node contains the value of its highest point. The quad-tree is traversed recursively while the ray height is below the highest point of a node. The ray is marched forward to the nearest intersecting node if a leaf node cannot be reached. Then, at a leaf node, an accurate intersection test is done.

Other GPU based methods have been used in [Qu03a], [Dic09a], [Luo12a] with limited hardware ray tracing support where an intersection between the viewing ray and the height field is calculated for each pixel on a fragment shader. The hybrid methods proposed in [Amm10a], [Dic10b] use ray-casting only when it actually performs better and switch back to other methods otherwise.

These ray-casting based methods have the advantages of concise memory requirements and steady frame rates being an the image order algorithm. However, they are not suitable for applications which use terrain geometry for other purposes like game engines and SVS. They suffer from pixel aliasing artifacts and other graphical glitches due to trace misses. There is also limited hardware support for ray tracing and in higher resolutions, they become too slow due to the high number of ray traces that must be done.

Triangulation methods pre-process the terrain and convert its height-field to polygons first and then render them using regular polygonal rendering techniques. The most trivial method is to generate a quad between any four adjacent samples form the height-field and this generates in high quality results but is extremely slow for large terrains. LOD algorithms are devised to pre-process the height-field before rendering and reduce it to the least possible number of geometrical objects to maintain very short response times. This is done by refining and coarsening the mesh based on pre-defined error metrics. Usually, some sort of multi-resolution representation of terrain data fitted to hierarchical data structures is used.

Wavelet based methods include the method used in [Fen12a] with a variation of the Haar transform for height-field decomposition into an intuitive multi-resolution representation. Different LODs are obtained by recursively averaging and differencing yielding and an approximation and a set of detail coefficients. The terrain representation can then be reconstructed to any LOD by applying the reverse process on lower-resolution versions where the simplest approximation is a single value representing the whole terrain. Wavelets are inherently multi-resolution data structures, have good compression ratio, and provide efficient means to store terrain data as the same amount of data is needed to represent all different resolutions combined. However, there are little benefits for using wavelets for terrains with high frequencies including mountains and peaks and there are considerable overhead related to the process of restoring the various LODs from the transform each time they are needed.

Losasso et at. in [Los04a] proposed a method which is called the geometry clipmaps. Terrain geometry is cached in a set of nested regular grids using a mipmap pyramid of L levels. A square window of samples is cached for each level. Each level is stored in a vertex buffer and a normal map and as the viewer moves, the clipmap windows are shifted and updated with new data. Due to its uniformity, this method can achieve steady frame rates but this also means that it is not adaptive as LOD selection is based only on the distance from the viewpoint not taking into account the actual approximation error introduced by this LOD. As a result, a flat part of the terrain will always be rendered with detailed meshes if it's close to the viewpoint and a curved part

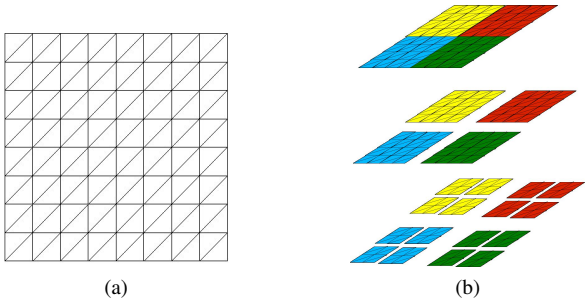
of the terrain will always be rendered with low detailed meshes if it's far from the viewpoint. This method also needs frequent CPU-GPU communications as a vertex buffer update must be done for every viewer motion and an index buffer update must be done per frame and many rendering calls have to be done per a single level due to the way it's designed. A GPU friendly adaptation of this method is proposed in [Asi05a] where each level is stored as an elevation map instead of a vertex buffer and constant vertex/index buffers are used for rendering all levels resulting in less CPU intervention and compact terrain representation.

Yalçin et al. in [Yal11a] used a diamond-based hierarchy to represent terrain. The hierarchy is generated by recursively applying the Longest Edge Bisection (LEB) operator to the pair of triangles decomposing the terrain height-field. Each diamond is subdivided by bisecting both of its triangles and the hierarchy can be modeled as a directed acyclic graph (DAG) where each resolution level is the set of diamonds at the same depth of the graph. The diamonds central vertices directly correspond to the grid points and the hierarchy is efficiently stored on the GPU as a simple array of error values of the same size as the original height-field. However, the process of generating the DAG is not simple as the entire terrain data set must be processed first.

Quad-trees have been used extensively for representing terrain (see [Paj02a], [Li12a]). Quad-tree based methods generally use a hierarchical quad-tree based data structure that is subdivided recursively until a pre-defined condition is met. This condition defines the accuracy of the resulting approximation of the terrain. If the condition is not met, the area is subdivided again into four quadrants and each quadrant is then re-evaluated. This allows for adaptive triangulation where the process of subdivision stops when the pre-defined condition is met regardless of the current depth of the quad-tree. This is a clear advantage over other uniform methods which could approximate the same terrain with a much larger number of polygons. However, this results in a variable frame rate that is based on terrain complexity. Also, all LODs must be available for every block of the terrain and if they are pre-calculated, memory requirements can be high and calculating them on-demand on the CPU introduces a major performance bottleneck.

3 OUR APPROACH

In this section, the proposed quad-tree based method for terrain rendering is presented. As Figure 1 shows, the terrain is logically seen as a quad-tree with each leaf consisting of a $k \times k$ terrain block which is the smallest unit of geometry that will be sent to the GPU for rendering. This method allows a single batch of geometry to be used for rendering any of the terrain blocks by



(a) (b)

Figure 1: A view of a 9×9 terrain: (a) as a mesh and (b) as a quad-tree with $k = 3$ terrain blocks represented as leaves.

varying the translation t and scaling s parameters sent to the GPU along with the block geometry as in [Asi05a], [Li12a]. The most detailed block has a scaling value of one and but lower resolution blocks have greater scaling values representing a larger area of the terrain with the same geometry. This lowers the required geometry GPU memory to a fixed value equal to the memory needed to represent only one block. The approximation errors due to switching to lower LODs are calculated entirely on the GPU and stored as a single texture. The CPU is then used to select the appropriate LOD for each block and to send rendering commands with the desired LOD and the block translation/scaling parameters.

Figure 2 shows an overview of the method. It works in two separate concurrent activities where each activity is composed of a CPU thread with an associated GPU context. The first activity is called the approximation errors activity and is responsible for building the quad-tree LOD hierarchy and the second activity is called the blocks rendering activity and is responsible for the final rendering process.

The approximation errors activity fits the height-field logically into a quad-tree hierarchy representing the different LODs as screen-space approximation error values in pixels that result from using a lower LOD. The calculation process is done on the GPU and the hierarchy is physically stored as a two dimensional GPU texture of the same size as the height-field and is then downloaded from the GPU and stored in main memory for later access by the blocks rendering entity. As the viewpoint changes, the approximation errors activity recalculates the error values based on the new viewpoint.

The blocks rendering entity accesses the texture generated by the approximation errors activity and traverses the represented quad-tree top-down starting from the center point of the texture and subdividing until the screen space error is below a pre-set threshold and then a block rendering request is sent directly to the GPU with the appropriate LOD scale and block translation values.

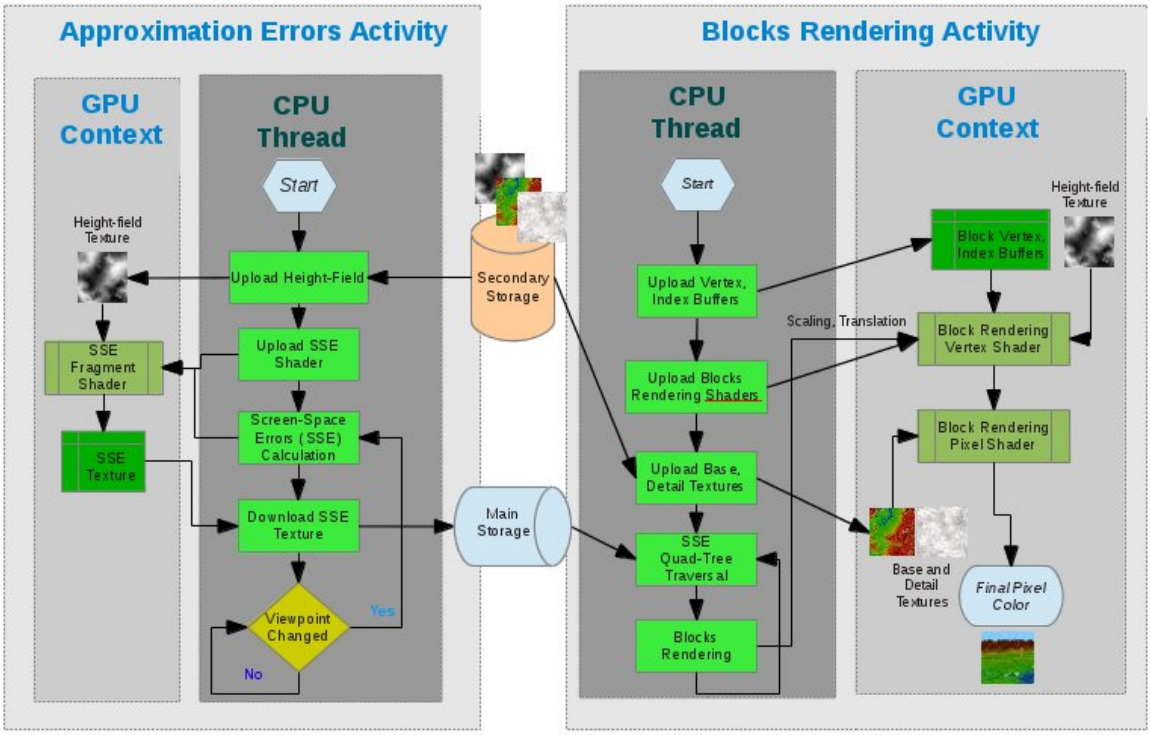
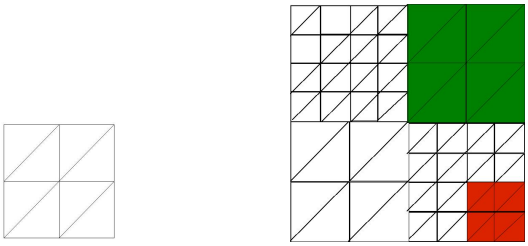


Figure 2: A general overview of the method

The following three subsections describe the details of this method.

3.1 Terrain Blocks



(a) (b)

Figure 3: (a) A single mesh with $k = 3$ in used for rendering all terrain blocks of (b) a terrain with 9×9 vertices. The red shaded area is of LOD 0 and is rendered with $s = 1$ and $t = (6,0)$ while the green shaded area is of LOD 1 and is rendered with $s = 2$ and $t = (4,4)$

The basic rendering primitive of the proposed method is the terrain block which is defined as a batch of geometry described by an index buffer and a vertex buffer representing the mesh shown in Figure 3a. The mesh uses $k \times k$ vertices and consists of $2(k-1)^2$ triangles where k is a parameter defining the granularity of the algorithm. A terrain block is rendered using a single drawing call so choosing a large value of k means less drawing calls but also limits the number of possible LODs and results in more triangles representing the same terrain since the terrain block geometry defines the minimum number of triangles required for representing any given area of the terrain regardless of its simplicity.

A terrain block can be translated and scaled to represent any area of the terrain at a single LOD by varying translation and scaling parameters sent to the fragment shader. Figure 3b shows how terrain blocks are used to represent 2×2 areas of the terrain at the most detailed LOD 0 using a scaling parameter of $s = 1$ and also to represent 4×4 areas of the terrain at a lower LOD 1 using a scaling parameter of $s = 2$. This allows sharing the same vertex buffer and index buffer among all of the terrain blocks covering the whole terrain and results in very concise GPU memory requirements for storing terrain geometry.

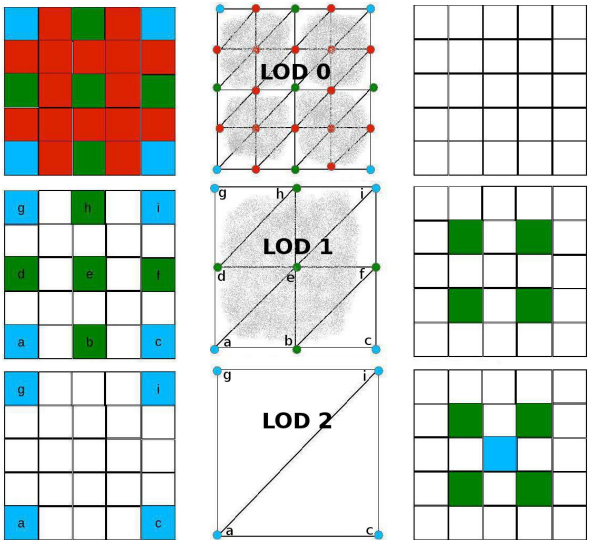
3.2 Approximation Errors Activity

The terrain data is represented by a height-field with each pixel corresponding to one vertex and where the location of the pixel defines the x and z coordinates of the vertex and the pixel value represents the height of that vertex as the 2D height function: $y = H_{x,z}$.

The height-field is simplified into multiple LODs by iteratively down-sampling to obtain lower detailed representations of the terrain starting with LOD 0. The number of vertices for a specific LOD is given by: $VC_i = (2^{n-i}+1)(2^{n-i}+1)$ where $i \geq 0$ defines the LOD level with $i = 0$ being the most detailed level and $i = n$ being the lowest detailed level and $n > 1$ is a value chosen that defines the height-field dimension. A sample

height-map of $n = 2$ and its triangulation for all possible LODs are shown in Figure 4a and Figure 4b respectively.

Figure 4 shows the down-sampling process for a terrain with $n = 2$ from LOD 0 to LOD 2. Each iteration starts with dividing the input height-field at a specific LOD into groups of 3×3 vertices and dropping vertices from each group merging adjacent groups at LOD i into only one group at LOD $i+1$ and then using these groups as the input of the next iteration until only 2×2 vertices are left no more groups can be formed meaning that no further down-sampling is possible as shown in Figure 4b.



(a) (b) (c)

Figure 4: Down-sampling process for: (a) a height-field with n=2, (b) its triangulation, (c) errors texture. Red, green and blue colors show which vertices exist in the different LODs 0, 1, 2 respectively from top to bottom.

Each group vertices are given the labels a,b,c,d,e,f,g,h,i as shown in Figure 4b. Vertices with labels b,d,e,f,h are dropped and each dropped vertex height value is interpolated to a new value using two of its neighbor vertices that exist at the new LOD defining the interpolated height function given by:

$$\begin{aligned} IH_{x_b,z_b} &= H_{x_a,z_a} + 0.5 \times (H_{x_c,z_c} - H_{x_a,z_a}) \\ IH_{x_d,z_d} &= H_{x_a,z_a} + 0.5 \times (H_{x_g,z_g} - H_{x_a,z_a}) \\ IH_{x_e,z_e} &= H_{x_a,z_a} + 0.5 \times (H_{x_i,z_i} - H_{x_a,z_a}) \\ IH_{x_f,z_f} &= H_{x_c,z_c} + 0.5 \times (H_{x_i,z_i} - H_{x_c,z_c}) \\ IH_{x_h,z_h} &= H_{x_g,z_g} + 0.5 \times (H_{x_i,z_i} - H_{x_g,z_g}) \end{aligned} \quad (1)$$

This introduces a vertex world-space interpolation error at each interpolated vertex and is given by $WIE_{x,z} = H_{x,z} - IH_{x,z}$. This error can be projected on the screen for the screen-space interpolation error

$$SIE_{x,z} = Project(H_{x,z} - IH_{x,z}) \quad (2)$$

The dropped vertex with the maximum vertex interpolation error is taken as the overall approximation error for the dropped vertices of this group and is given by:

$$OE = max(SIE_{x_b,z_b}, SIE_{x_d,z_d}, SIE_{x_e,z_e}, SIE_{x_f,z_f}, SIE_{x_h,z_h}) \quad (3)$$

A new texture of the same dimensions as the height-field texture is created to store the screen-space approximation errors. This texture is filled progressively during the down-sampling process starting from LOD 0 where each iteration adds more data to the texture as shown in Figure 4c.

The group j interpolation error for LOD i can then be calculated as the maximum of the dropped vertices error for group j at LOD i and the maximum of each group interpolation error of the groups which formed this group in the previous iteration at LOD $i-1$. This error can be seen as the maximum screen-space error that can result from using the simplified version of this group at this LOD instead of the original vertices at LOD 0 and is stored in the errors texture at the position of the dropped center pixel in this group labeled e which is used again for the next iteration.

If k,l,m,n are the groups that formed the group j in the previous iteration (if any) then group j interpolation error for LOD i is given by:

$$GIE_{i,,j} = \begin{cases} 0 & i = 0 \\ max(OE_{i,j}, GIE_{i-1,k}, GIE_{i-1,l}, & \\ GIE_{i-1,m}, GIE_{i-1,n}) & i > 0 \end{cases} \quad (4)$$

Figure 4c shows the process of filling the errors texture. Initially, the error texture is all zeros because at LOD 0 no vertices are dropped and $GIE_{0,,j} = 0$ for all groups. At LOD 1, the four groups are merged and their $GIE_{1,,j}$ value is calculated and stored at the center points which are green shaded in the figure. At the next and last iteration of LOD 2, the $GIE_{2,0}$ value of the single group which were formed from the previous iteration is calculated based on the previous $GIE_{1,,j}$ values and stored in the center point which is blue shaded forming the final errors texture which only has non-zero values at the four center points of LOD 1 groups and the center point of the single LOD 2 group.

The process of building a quad-tree of the approximation errors representing the different LODs is done using the approximation errors activity. As the viewpoint moves, the activity CPU thread initiates the approximation errors process by iteratively sending a simple quad to the activity offscreen GPU rendering buffer context for each LOD starting with LOD 1 (in LOD 0, approximation errors are zero) which has been uploaded previously with a fragment shader (see Algorithm 1) that includes the logic of calculating screen-space errors $GIE_{i,,j}$ for each group based on the height-field texture

if fragment is not a group center for LOD i **then**
  return discard fragment
for all v in dropped vertices (b, d, e, f, h) **do**
  $H_{x_v,,z_v} = texture2D(heightField, texcoord_{x_v,z_v})$
  Calculate IH_{x_v,z_v}, SIE_{x_v,z_v} (Equations 1,2)
Calculate $OE_{i,j}$ (Equation 3)
for all g in LOD i-1 groups (k, l, m, n) **do**
  $GIE_{i-1,,g} = texture2D(errorsTexture, x_g, y_g)$
Calculate $GIE_{i,,j}$ (Equation 4)
return $GIE_{i,,j}$

Algorithm 1: Pseudocode for errors fragment shader

passed by the CPU during initialization and its own errors texture from the previous iteration using the above formulas and storing the error as the texture value of the group center coordinates in an errors texture of the same dimensions as the height-field.

The root value of the built quad-tree is the value of the center point of the errors texture which is the center point of the single group at the lowest detailed LOD representing the overall screen-space error of using that group for representing the terrain. Its four children are the values of the center points of the four groups that were originally merged to form that group and each one of them in turn can be subdivided again into four groups until the leafs of the quad-tree are found and they represent the original groups at LOD 0. Figure 5 shows the built quad-tree for a terrain height-map with $n = 2$.

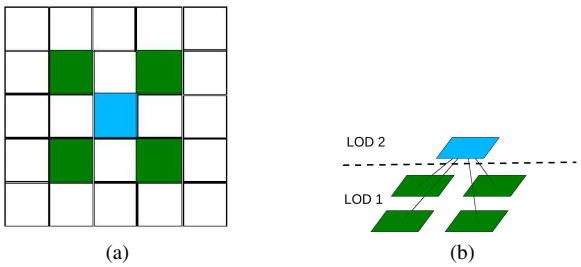


Figure 5: (a) The errors texture of a height-map with $n = 2$ and (b) the built quad-tree from it.

The errors texture is finally downloaded from the GPU and stored as a 2-dimensional array on the main memory for use by the blocks rendering activity, which is detailed in the next subsection, to render the terrain based on the built quad-tree of screen-space errors.

3.3 Blocks Rendering Activity

The terrain blocks rendering activity takes as input the quad-tree built using the approximation errors activity. The quad-tree is traversed top-down using the errors texture fetched earlier from the GPU starting from the center point which represents the screen-space error of using the just four vertices to represent the whole terrain. Since the basic unit for rendering is the terrain block mesh which consists of $k \times k$ vertices, the actual screen-space error of using a terrain block at a given LOD i is fetched by recursively traversing the quad-tree top-down starting from the center point of the terrain block until the number of leafs, which represent center points for different groups, is exactly equal to $(k-1)(k-1)$ and then their maximum value is taken as the terrain block b error at LOD i and is given by $TBE_{i,b} = max(GIE_{i,1}, GIE_{i,2}, \ldots, GIE_{i,(k-1)\times(k-1)})$

The process starts on the CPU thread associated with the blocks rendering activity by fetching from the quad-tree the scree-space error of using the least detailed possible LOD n which approximates the terrain using a single terrain block covering the whole terrain. If that error is below a pre-set threshold value δ that defines the maximum tolerable screen-space error, the terrain is not subdivided and the CPU thread marks that single terrain block to be rendered by the GPU context associated with the blocks rendering activity.

If the error is greater than δ, the quad-tree is recursively traversed top-down one level to reach the next detailed LOD $n-1$ subdividing the terrain block into four terrain blocks each located at a different quadrant of the terrain. Screen-space errors for using the new four terrain blocks are fetched and compared to δ again and if any terrain block passes the test, it's marked to be rendered or else subdivided again into four new blocks.

This process continues until all leafs pass the screen-space error test or a given terrain block cannot be subdivided anymore because it's already at the most detailed LOD 0 and in which case it's marked to be rendered too. An intersection test is also done against the current camera view frustum to cull invisible terrain blocks before subdividing it or marking it for rendering. The subdivision stopping condition for a block b at LOD i can then be formulated as $TBE_{i,b} \leq \delta$ where δ is a pre-set threshold for screen-space errors in pixels.

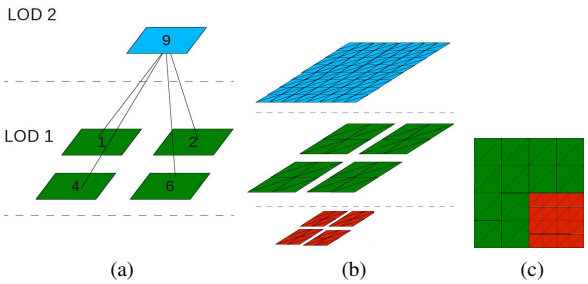


Figure 6: Traversal process for (a) the errors quad-tree and (b) its associated terrain blocks tree to reach at (c) the final terrain blocks to send to the GPU

Figure 6 shows the quad-tree traversal process for rendering a terrain with $n = 2$, $k = 2$ and $\delta = 5$. The traversal starts with the root of the errors quad-tree in Figure 6a which is blue shaded and has the value 9. This value is greater than the threshold $\delta = 5$ and therefore, the corresponding single blue shaded terrain block shown

in Figure 6b for LOD 2 cannot be used to approximate the terrain and the quad-tree is traversed to the next level where the four green shaded terrain blocks are evaluated for use instead of the blue shaded terrain block. The evaluation process continues by comparing the values of the four leaves of the errors quad-tree against $\delta = 5$. Three of the green-shaded leaves pass the test as their values are $1,2,4$ and they are less than 5 so the corresponding three terrain blocks shown in Figure 6b are used for LOD 1. However, the fourth value is 6 which is greater than 5 and as such the corresponding terrain block is subdivided into four new terrain blocks at LOD 0 which is the most detailed LOD. The result is shown in Figure 6c which uses four terrain blocks at LOD 0 (red shaded) and three terrain blocks at LOD 1 (green shaded) to render the terrain.

Each terrain block marked for rendering is sent to the GPU context for rendering along with translation and scaling values that define the area of terrain that is covered by it.

For each terrain block b at LOD $i+1$ subdivided into children terrain blocks c,d,e,f at LOD i, the new scaling value for children blocks is given by $s_i = \frac{2^i}{k-1}$ and the new translation values for them are given by:

$$\begin{aligned} t_{i,c} &= t_{i-1,b} \\ t_{i,d} &= t_{i-1,b} + (2^i, 0) \\ t_{i,e} &= t_{i-1,b} + (0, 2^i) \\ t_{i,f} &= t_{i-1,b} + (2^i, 2^i) \end{aligned} \tag{5}$$

where the root terrain block b_{root} covering the whole terrain at the least detailed LOD n has the scaling and translation parameters $s_{n,b_{root}} = \frac{2^n}{k-1}$ and $t_{n,b_{root}} = (0,0)$. The rendering is done using a single shared vertex buffer and a single index buffer that are uploaded to the GPU once along with required vertex and fragment shaders. These buffers define the triangulation shown in Figure 1a using $(k-1)(k-1)$ vertices. Each terrain block is rendered by rendering the triangulation shown in Figure 1a after transforming the vertices original x,z values into final tranfromed values x',z' by the passed translation and scaling parameters in a vertex shader as follows:

$$\begin{aligned} x' &= x \times s_i + t_i \\ z' &= z \times s_i + t_i \end{aligned} \tag{6}$$

The component y which defines the height of the vertex is fetched on the vertex shader by sampling the height-field which is passed to the shader as a texture at the texture coordinates directly calculated from x',z' after converting to the $[0,1]$ range by dividing by the height-field dimension 2^n+1:

$$texcoord_{base} = (\frac{x'}{2^n+1}, \frac{z'}{2^n+1}) \tag{7}$$

$$y' = texture2D(heightfield, texcoord_{base}) \tag{8}$$

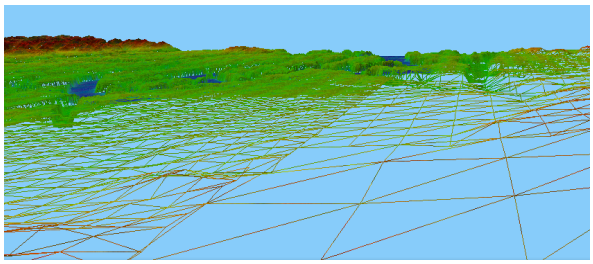


Figure 7: The Puget Sound dataset rendered using the proposed method in textured wire-frame mode

3.4 Terrain Texturing

Texturing the terrain is done by passing the input terrain base texture to a fragment shader which samples it to calculate the base pixel color c_{base} using the same texture coordinates $texcoord_{base}$ used to sample the height-field because the terrain base texture is of the same dimensions as the height-field as follows:

$$c_{base} = texture2D(basetexture, texcoord_{base}) \tag{9}$$

A small detail texture is also passed to the fragment shader to make up for the missing details in the base texture due to the small pixel-to-texel ratio. The second set of texture coordinates used for sampling the detail texture is calculated as $texcoord_{detail} = DTTF \times texcoord_{base}$ where $DTTF$, the detail texture coordinates tiling factor, is a factor describing the tiling property of the detail texture. The larger this factor, the better pixel-to-texel ratio but also the more visible tiling repetition artifacts. The color coming from the detail texture is then:

$$c_{detail} = texture2D(detailtexture, texcoord_{detail}) \tag{10}$$

The fragment shader then calculates the final pixel color c_{final} by blending the values sampled from the detail texture c_{detail} and the base texture c_{base} using:

$$c_{final} = \alpha \times c_{base} + (1-\alpha) \times c_{detail} \tag{11}$$

where α is a weight in the range $[0,1]$ given to the color fetched from the base texture where a value of $\alpha = 1$ means that the detail texture will be ignored.

4 RESULTS

We used the Puget Sound dataset (see [Usg01a]) to test the proposed methodology. It consists of 4097×4097 16-bit samples with each pixel unit (0 to 65535) corresponding to 0.1 meter and with inter-pixel spacing of 40 meters. The test was performed on a Linux machine with an Intel(R) Core(TM) i5 CPU M 430 @ 2.27GHz processor with 4GB RAM and an nVIDIA GeForce GT 330M graphics card. The parameter $n = 12$ is chosen to match the dimensions of the height-field and a terrain block is chosen to be of 33×33 vertices which means that $k = 33$.

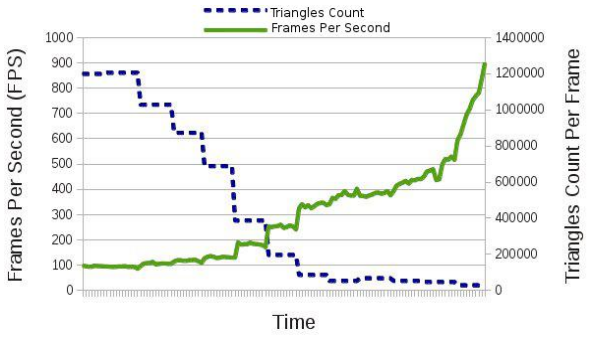


Figure 8: A plot of the frame rate and triangles count against the experiment time with $\delta = 1$

The test is a fly-over above the terrain starting exactly from the center of one edge of the terrain and flying forward to the other edge by a speed of 3600 km/h. The output is projected into a window of 640×480 pixels. Figure 8 shows a plot of the frame rate (FPS) and the triangles count per frame at a maximum screen-space error of $\delta = 1$ pixel.

At the start of the test, the frame rate is 97 FPS and the triangle count is 1198080 per frame. This is because all of the terrain is visible and the camera is close to large parts of it so the error for using lower LODs was larger than $\delta = 1$ for these parts. As the camera moves forward to the center of the terrain which is clear in Figure 8 from the intersection point between the two curves, the triangle count drops down to 300000 triangles raising the frame rate to about 200 FPS. This is expected because half of the terrain has already been culled and only some of the remaining visible parts need to be rendered using higher LODs. At the end of the flight at the other edge, the camera sees only a small part of the terrain which explains the low triangles count and the very high frame rate. Our method, for this test, achieved an average frame rate of 266 frames per second and as shown in Figure 8, it was able to reach a triangle throughput of $97 \times 1198080 = 120M\triangle$/sec.

For comparison with other publications work, their reported results are normalized based on a public graphics cards benchmarks database available online at [Vid14a] and the expected results on the testing machine are listed in Table 1 along with the actual result of the proposed methodology.

Publication	**$\triangle$ Throughput**
Feng et al. in [Fen12a]	53 $M\triangle/sec$
Yalçin et al. in [Yal11a]	110 $M\triangle/sec$
Asirvatham et al. in [Asi05a]	71 $M\triangle/sec$
Li et al. in [Li12a]	45 $M\triangle/sec$
Livny et al. in [Liv09a]	97 $M\triangle/sec$
Proposed Method	120 $M\triangle/sec$

Table 1: Comparison with estimated hardware normalized average $\triangle$ throughput for selected publications.

The higher triangle throughput reported in [Yal11a] is due to the large number of triangles per batch as Yalçin et al. reported 10 frames per seconds in [Yal11a] for the same dataset with a static mesh using a graphics card that scored three times more than the graphics card used for performing this test. This is expected because they didn't use any culling or LOD technique but focused instead on allowing dynamic changes on the terrain while rendering. This is not directly possible with our proposed method.

Livny et al. in [Liv09a] reported results that are close to the proposed method but they implemented an out-of-core technique for handling distant patches and they use more geometry because of the way they handle cracks. As a result of the proposed GPU-resident quad-tree LOD construction code, the proposed method outperformed the method used in [Li12a] despite the fact that the two methods use constant buffers for rendering.

From Table 1, it can be seen that the proposed method is comparable to recent publications but unlike [Asi05a], [Li12a] and [Fen12a] where LOD selection is based on some distance criteria or [Liv09a] where it is based on an arbitrary precision factor, the proposed method can guarantee a maximum screen-space error of δ pixels.

Since all terrain blocks are rendered using a single constant small mesh, the GPU memory used for geometry is constant and is equal to the storage used by the vertex buffer and the index buffer. Each vertex consists of three 32-bit floating-point components for x, y, z of 4 bytes each and the vertex buffer consists of kxk vertices so the total vertex buffer size is $VB_{size} = k \times k \times 3 \times 4 = 65 \times 65 \times 3 \times 4 = 50700$ bytes. The index buffer consists of 32-bit indices with every triangle defined by three indices. The number of triangles in the mesh is $T_{count} = 2(k-1)(k-1) = 8192$ triangle and the indices count is $IB_{count} = 3 \times T_{count} = 24576$ index. Thus the total index buffer size is $IB_{size} = 4 \times IB_{count} = 98304$ bytes.

The used GPU memory for geometry is then $VB_{size} + IB_{size} = 149004$ bytes. This shows that even very large terrain data sets can be rendered with the same GPU geometry memory footprint. By using texture compression for the height-field and the base texture, the GPU texture memory could also be reduced substantially.

5 CONCLUSION

We presented a method for real-time terrain rendering using multiple CPU threads that drive multiple GPU contexts separately. The CPU overhead is kept minimum as the GPU does most of the work. *The main contribution of this work is the novel GPU-resident errors quad-tree construction process that is performed on the GPU by a fragment shader.* The CPU is then used to download the errors quad-tree as a simple GPU texture and use it again to drive the blocks rendering

activity GPU context by sending it only simple translation and scaling parameters per block. These parameters are used by a vertex shader to reconstruct the intended terrain block by using a shared vertex and index buffers and calculating the vertices height values on-the-fly from the height-field which has been previously uploaded as a GPU texture. This cuts down the GPU memory requirements for storing terrain geometry as only a constant small vertex and index buffer is stored in addition to the height-field.

The results of the proposed method for the tested dataset outperform similar methods when taking LOD accuracy and frame rates into accounts. However, the process of downloading the errors quad-tree from the GPU as a texture is considered as a bottleneck in this method. This can be overcome by: distributing the process among more different frames, using only one byte per pixel for the errors texture, and finally there is no need to downloading the whole errors texture as this can be done on-the-fly while traversing the errors quad-tree by first downloading the root and then downloading only when a subdivision is required.

In the future, we will look into improving this method by moving the quad-tree traversal code into the GPU so that no texture downloading is required to be done by the CPU. We hope to fix the visual cracking and popping artifacts that result from stitching different LODs and we will also look into dynamically splitting and compressing the height-field and base textures to handle larger terrain datasets and to cut down GPU texture memory requirements.

6 REFERENCES

[Amm10a] Ammann, L., Génevaux, O., and Dischler, J. "Hybrid rendering of dynamic heightfields using ray-casting and mesh rasterization." Proceedings of Graphics Interface 2010. Canadian Information Processing Society, 2010.

[Asi05a] Asirvatham, A. and Hoppe H. "Terrain rendering using GPU-based geometry clipmaps." GPU Gems 2, pages 27-46, 2005.

[Coh93a] Cohen, D. and Shaked, A. Photo-Realistic Imaging of Digital Terrains. Computer Graphics Forum, Vol 12, pages 363-373, 1993.

[Coh96b] Cohen-Or, D., Rich, E. et al. "A real-time photo-realistic visual flythrough." IEEE Transactions on Visualization and Computer Graphics, Vol 2, Issue 3, pages 255-265, 1996.

[Dic09a] Dick, C., Krüeger, J., and Westermann, R. "GPU ray-casting for scalable terrain rendering." Proceedings of EUROGRAPHICS. Vol. 50. 2009.

[Dic10b] Dick, C., Krüeger, J., and Westermann, R. "GPU-aware hybrid terrain rendering." Proceedings of IADIS Computer Graphics, Visualization, Computer Vision and Image Processing '10, pages 3-10, 2010.

[Dun79a] Dungan Jr, W. "A terrain and cloud computer image generation model." ACM SIGGRAPH Computer Graphics. Vol. 13. No. 2. ACM, 1979.

[Fen12a] Feng, Y., Wang, R. et al. "Real-time Rendering Techniques of Lunar Massive Terrain Data Based on Wavelet." Journal of Computational Information Systems, Vol 8(14), pages 6079-6085, 2012.

[Hen04a] Henning, C., and Stephenson, P. "Accelerating the ray tracing of height fields." Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia. ACM, 2004.

[Li12a] Li, Q., Wang, X. et al. "Real-Time Visualization for Large Scale Terrain Based on Linear Quadtree." AsiaSim '12. Springer Berlin Heidelberg, pages 331-339, 2012.

[Liv09a] Livny, Y., Kogan, Z., and and El-Sana, J. "Seamless patches for GPU-based terrain rendering." Proceedings of International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision '07, pages 201-208, 2007.

[Los04a] Losasso, F. and Hoppe, H. "Geometry clipmaps: terrain rendering using nested regular grids." ACM Transactions on Graphics (TOG), Vol 23, Issue 3, pages 769-776, 2004.

[Luo12a] Luo, J., Ni, G., et al. "Quad-tree atlas ray casting: a gpu based framework for terrain visualization and its applications." Transactions on Edutainment VII. Springer Berlin Heidelberg, pages 74-85, 2012.

[Paj02a] Pajarola, R. "Overview of quadtree-based terrain triangulation and visualization". Department of Information & Computer Science, University of California, Irvine, 2002.

[Qu03a] Qu, H., Qiu, F., et al. "Ray tracing height fields." Proceedings of Computer Graphics International '03. IEEE. pages 202-207, 2003.

[Usg01a] U.S.G.S., THE UNIVERSITY OF WASHINGTON: Puget sound terrain dataset. http://www.cc.gatech.edu/projects/large_models/ps.html.

[Vid14a] Video card benchmarks database. http://www.videocardbenchmark.net.

[Yal11a] Yalçin, M., Weiss, K., and De Floriani, L. "GPU algorithms for diamond-based multiresolution terrain processing." Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization. Eurographics Association, pages 121-130, 2011.