PAPER

Geometry Splitting: An Acceleration Technique of Quadtree-Based Terrain Rendering Using GPU

Eun-Seok LEE[†a)], *Nonmember and* **Byeong-Seok SHIN**[†b)], *Member*

SUMMARY In terrain visualization, the quadtree is the most frequently used data structure for progressive mesh generation. The quadtree provides an efficient level of detail selection and view frustum culling. However, most applications using quadtrees are performed on the CPU, because the pointer and recursive operation in hierarchical data structure cannot be manipulated in a programmable rendering pipeline. We present a quadtree-based terrain rendering method for GPU (Graphics Processing Unit) execution that uses vertex splitting and triangle splitting. Vertex splitting supports a level of detail selection, and triangle splitting is used for crack removal. This method offers higher performance than previous CPU-based quadtree methods, without loss of image quality. We can then use the CPU for other computations while rendering the terrain using only the GPU.

key words: *hierarchial data structure, terrain rendering, level of detail, real-time rendering, quadtree*

1. Introduction

Terrain visualization methods have been used in many applications, such as video games and flight simulations, for representing outdoor scenes. In general, because huge datasets are used in terrain visualization, much memory space and computation time is required. Therefore, we have to apply some mesh simplification methods while maintaining image quality.

A number of researchers have proposed the CLOD (Continuous Level Of Detail) methods for rendering the large terrain datasets in real time. The CLOD methods using hierarchical data structures, such as quadtree [1]–[5], triangle binary tree [6]–[8], longest edge bisection [9], right triangle hierarchies [10], [11], texture hierarchies [12]–[15] and vertex hierarchies [16] are usually executed by the CPU. These methods simplify the geometry efficiently by using CLOD and VFC (View Frustum Culling).

In recent years, the performance of graphic hardware has continually improved. A GPU provides much faster graphic operations and parallel computations, compared to a CPU. However, there are the bottlenecks between CPU and video memory. To overcome the communication bottleneck, use of a geometry cache has been suggested [17]–[19]. Yusov proposed a quadtree-based progressive rendering method which uses optimized patches for GPU [20]. This method provides a balanced terrain patches by caching the patch geometry in GPU's memory. The patches will be used across the several scenes. However, since the size of video memory is limited, a large data requirement may introduce a communication overhead between the CPU and the video memory. To decrease this overhead, several data compression methods have been presented [21]–[23]. Dick proposed a geometry compression and decoding method using the GPU [24]. This compresses the preprocessed geometric data using the CPU and decodes them in the GPU's rendering pipeline using a geometry shader [25]. However, these methods struggle because of limited cache memory. To alleviate this problem, Schneider et al. proposed the progressive transmission of geometry to decrease the communication time [26]. Livny et al. suggested the use of seamless patches for reducing the communication between CPU and GPU [27]. The predefined patches are stored in a cache and stitched between triangular tiles that have different LOD levels.

Geometry clipmap techniques [28]–[32] can render large terrain datasets by using a clipmap [33]. However, the LOD level is selected in a world space based on several quadrilateral regions. Therefore, geometry popping may occur, caused by inaccurate CLOD computation.

A quadtree [34] which is a hierarchical data structure, is widely used in terrain rendering. The quadtree-based triangulation methods [1]–[5] can provide higher-quality terrain images easily by the simple error metrics. However, quadtree cannot be handled by the GPU's rendering pipeline [35]. Therefore, previous methods involving quadtrees used only CPU-based execution.

A GPU-based LOD method using an STA (Seamless Texture Atlas) [36] has been suggested, which uses geometry images [37]. Niski et al. proposed a multi-grained LOD for load balancing between CPU and GPU using an STA [38]. In this technique, the CPU controls the LOD by using a quadtree, and the GPU manipulates detail levels using a hierarchical STA. To increase usage of the GPU, Chang and Shin proposed the ef-buffer [39]. This method performs most of quadtree-based computation in the GPU's rendering pipeline. However, crack removal is still performed on the CPU.

We present a fully GPU quadtree-based terrain-rendering method that represents the quadtree appropriately for the GPU's rendering pipeline. Figure 1 shows the overall procedure for our method. The input datum is a height field of size $n \times n$ ($n = 2d + 1$, where d is the depth of the quadtree). In a preprocessing step, we generate a quadtree texture, which stores the surface roughness values for effi-

Manuscript received October 19, 2009.
Manuscript revised August 13, 2010.

[†]The authors are with MediaLab, Dept. Computer Science and Information Engineering, Inha University, Inchon 402–751, Korea.
a) E-mail: elflee77@inha.edu
b) E-mail: bsshin@inha.ac.kr

DOI: 10.1587/transinf.E94.D.137

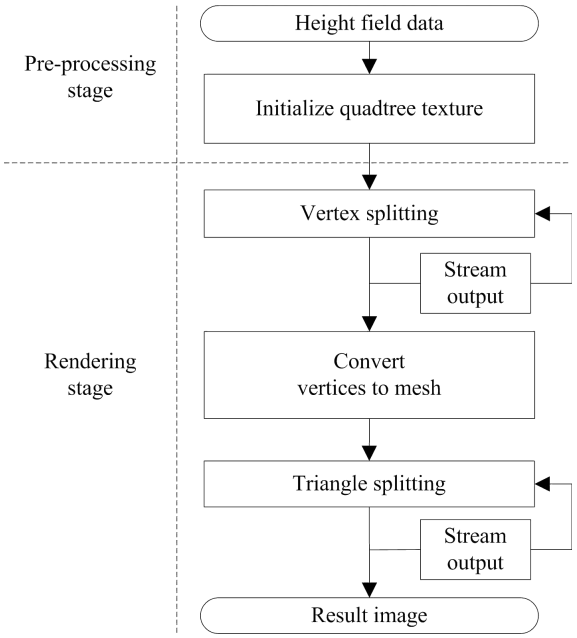


Fig. 1 Rendering procedure of our method.

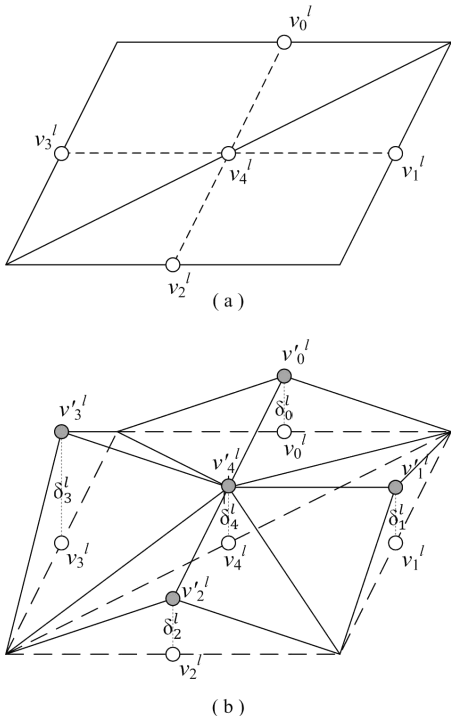


Fig. 2 Computation of geometric errors. (a) shows the vertices included in a quadrilateral block. They are located at the midpoints of each triangle's edge. (b) depicts the five geometric errors $\delta_0^l \sim \delta_4^l$, which are the distances between the midpoints of edges of the upper level block and vertices of the lower level block.

cient LOD control. In the rendering step, we calculate the detail level for each vertex, using the values of quadtree texture. When a more detailed level for a vertex is required in the LOD selection step, we split the vertex using the vertex-splitting method. This step is repeated d times, with the stream output buffer feeding back its output to the previous step as input data. After the vertex splitting, each vertex is converted into a quadrilateral block to generate the terrain mesh. However, adjacent blocks having different levels will have cracks caused by T-vertices. We use triangle splitting to handle this T-junction problem, and the cracks are removed.

In Sect. 2, we explain the main algorithms of fully GPU quadtree-based rendering in detail. The acceleration in our methods is presented in Sect. 3, and experimental results are given in Sect. 4. Finally, Sect. 5 concludes the paper.

2. Terrain Visualization Using GPU Quadtree

In this section, we present a GPU-based rendering method for quadtree-based terrain. In conventional quadtree-based methods, major procedures, such as LOD selection and VFC, are executed by the CPU. These require much computation and the rendering speed is slower than that for other GPU-based methods. Our method provides high performance without loss of image quality, because it performs the entire process in the GPU.

The GPU's rendering pipeline does not support pointer and recursive operations. Therefore, hierarchical data structures cannot be manipulated in the GPU. We suggest vertex splitting and triangle splitting as alternatives to pointer operations. These are implemented in DirectX10 Shader Model 4.0 [25] Recursive operations can be substituted by the stream output stage.

2.1 LOD Selection

In this section, we propose an efficient CLOD method that reduces geometry popping by using the screen space error of a vertex and the distance between a viewpoint and the terrain surface.

In conventional quadtree-based CLOD methods, geometry popping may occur, caused by geometric errors. The geometric error value δ can be computed as the distance between the midpoint of an edge in an upper level block v and the vertex in a lower level block v', as shown in Fig. 2. A block contains five geometric error values at its center and the midpoints of its four edges.

The surface roughness value σ, which is the maximum of the geometric errors, is used to compute the screen space error $\overline{\sigma}$. Figure 3 shows the evaluation of a screen space error value. In evaluation of the screen space errors, all δs of a block will be replaced into σ. Then we project all vs and v's onto the view plane using the world, view and perspective projection matrix. We obtain the screen space error values by evaluating the errors between pairs of projected vertices $\overline{v}$ and $\overline{v}'$.

Our method uses a threshold τ, representing the tolerable screen space error, as specified by the user. Using τ, we can compute a flag value f as shown in Eq. (1). When the maximum screen space error exceeds τ, $f = 1$ and the LOD selection is continued for lower level blocks. Other-

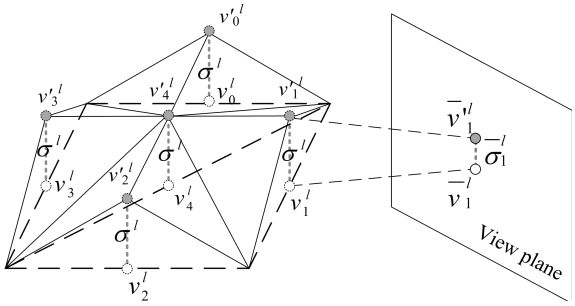


Fig. 3 Evaluations of the surface roughness value in screen space.

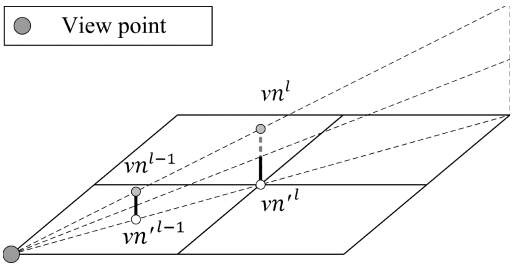


Fig. 4 A parent node's surface roughness value must be double that of its child node.

wise, LOD selection stops at that level.

$$f = \begin{cases} 1 & \text{if } \tau \le max(\overline{\sigma}_k^l) \\ 0 & \text{if } \tau > max(\overline{\sigma}_k^l) \end{cases} \quad \text{where } k \in [0,4] \qquad (1)$$

However, lower level blocks may have bigger geometric errors, causing incorrect LOD selections. Suppose that a node and its parent node have the same σ values. When a vertex v of a node is closer to the viewpoint than that of the parent's node, the screen space error of the node becomes bigger than that of its parent. Therefore, LOD selection can terminate at the parent node, even though the child node has the bigger screen space error. This incorrect LOD selection may cause geometry popping. To eliminate these incorrect LOD selections, we have to consider the distance of blocks from the view position. We use the boundary sphere P that covers a block. C_P is the center point and R_P is the radius of P. The LOD selection flag f can be rewritten as in Eq. (2), which determines if traversing the lower level blocks is required. If the viewing position V is inside the boundary sphere, while the surface roughness value is not zero, f has the value of 1.

$$f = \begin{cases} 1 & \text{if } |V, C_P| - R_P < 0, \sigma \neq 0 \\ & \quad \text{or } |V, C_P| - R_P \ge 0, \tau \le max(\overline{\sigma}_k^l) \\ 0 & \text{otherwize} \end{cases} \quad \text{where } k \in [0,4] \qquad (2)$$

Now we have to consider the maximum screen space error when V is outside of P. In Fig. 4, the dotted line shows the difference between the screen space errors of a block and its child blocks that have the same σ values. We can compute the maximum distance by locating V at the nearest vertex of the block in contact with P. Then we can evaluate the maximum difference, which is the same as the surface roughness value of adjacent child blocks. Therefore, we have to use a surface roughness value double that of the child nodes as the block's surface roughness value, if it is bigger than the block's other geometric error values. This may avoid incorrect LOD selection. Therefore, the surface roughness values are computed as in Eq. (3).

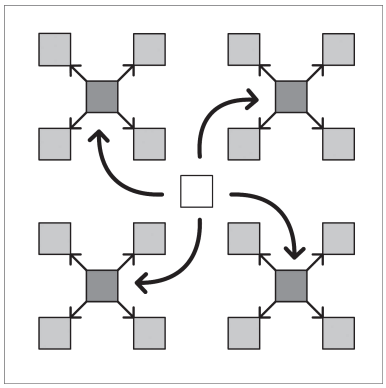


Fig. 5 An example of a quadtree texture for which $d = 3$. The white texel is the root node, with the others being its lower level texels.

$$\sigma^l = \begin{cases} max(\delta_k^l) & \text{if } l = d \\ max(max(\delta_k^l, \delta^{l+1} \times 2)) & \text{if } l < d \end{cases} \quad \text{where } k \in [0,4] \qquad (3)$$

2.2 Initialization of Quadtree Texture

A quadtree texture is the texture used for LOD selections in geometry shader. The resolution of quadtree texture is same as the resolution of height field data. It stores surface roughness values of each block. Therefore, the geometry shader can access the vertices surface roughness values by using their positions as the texture coordinates.

It does not have to be initialized more than once because the height field dataset does not change in general. This texture will be generated and uploaded in the GPU's video memory once in the pre-processing step.

Figure 5 depicts an example of a quadtree texture. The root node is located at the center of the texture. The texture is subdivided into four square blocks. Then the children of the root node are stored in the center of the divided blocks. By continuing these steps d times recursively, the quadtree texture will be completed.

By using the quadtree texture, LOD selection can be implemented in the GPU. We can approach the surface roughness value in the quadtree texture by the position of the block's center points.

2.3 Vertex Splitting

Instead of using a quadtree, we will exploit vertex splitting, which substitutes the pointers and recursive operations. Vertex splitting is performed in the GPU's rendering pipeline

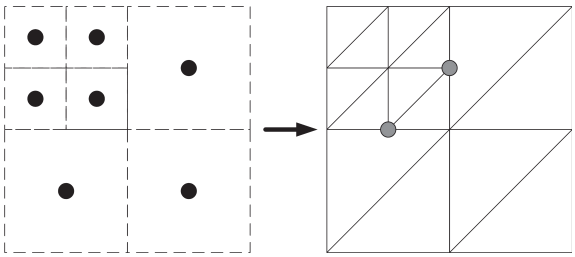


Fig. 6 Converting the vertices into quadrilateral blocks. The gray points are the T-vertices causing cracks.

using a geometry shader and the stream output stage.

In the geometry shader stage, we have a single primitive (a point, a line segment, or a triangle, with or without adjacency information) as its input. This stage can modify or delete primitives as well as create new primitives. The computation will be performed in parallel for each primitive. The stream output stage supplies the vertex data from the geometry shader stage to one or more buffers in video memory.

In the vertex-splitting method, the geometry shader inputs a single vertex, representing the root node of the quadtree, as an initial value (called the root vertex). The root vertex is located at the center of the root block. If we have to search for child nodes according to the value of the flag f, the geometry shader splits the vertex into four child vertices. These vertices are located at the centers of the sub-blocks divided from the root block. In this step, vertices will be produced and stored in the buffer in the video memory by the stream output stage. These vertices will then be fed back into the geometry shader and the process repeated d times. As a result, a group of vertices representing the subdivided blocks generated by LOD selection is produced.

The geometry shader performs these operations for each vertex in parallel. Therefore, this vertex-splitting process is faster than the previous quadtree traversal.

2.4 Triangle Splitting for Crack Removal

After the vertex-splitting step, the geometry shader converts the vertices into quadrangle blocks, as depicted in Fig. 6. These blocks make up the terrain mesh. However, cracks may appear at the gaps between blocks at different levels.

In the previous method [39], the crack removal process was performed by the CPU. We propose to use triangle splitting, a crack removal method, in the GPU's rendering pipeline, using a geometry shader and the stream output stages.

To remove cracks using the GPU, we have to determine which blocks cause cracks. When blocks adjacent to a specific block are at different levels, cracks may occur. To evaluate an adjacent block's level, we generate the four center points of neighboring blocks that have the same level as the target block, as shown in Fig. 7. We call these the *neighbor vertices*. Because one or more neighbor vertices of a block can be split in the vertex-splitting method, we split the block

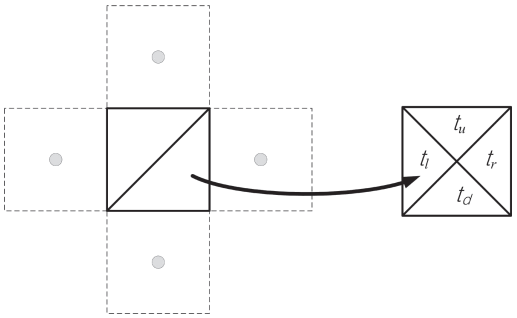


Fig. 7 If the neighboring block is at a lower level, the block is divided into four triangles. The gray vertex is the neighbor vertex of the block, which is divided into four triangles.

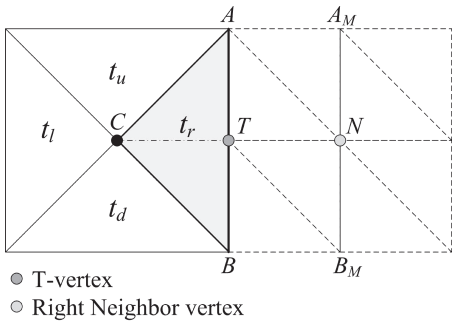


Fig. 8 The triangle-splitting method is performed in $\triangle ABC$ if the LOD flag f of the neighbor vertex N is 1.

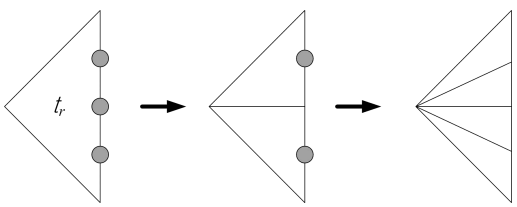


Fig. 9 Crack removal using triangle splitting. The gray vertices are the T-vertices.

into four triangles using the geometry shader. These four triangles will be used for crack removal in each direction (up, down, left, and right).

Triangle splitting, a parallel T-junction method, may be accomplished only in those blocks that we have divided into four triangles, as shown in Fig. 8.

We calculate the position of each triangle's neighbor vertex. In Fig. 8, the $\triangle ABC$ is the t_r of Fig. 7. It finds the neighbor vertex N. If N's flag is 1, we find that there is a T-vertex T at the center of $\overline{AB}$. To remove the crack, we have to split $\triangle ABC$ into $\triangle ATC$ and $\triangle TBC$. These steps will be performed recursively in the geometry shader $d-1$ times, using the stream output stage as in vertex splitting. The neighbor vertex of $\triangle ATC$ and $\triangle TBC$ can be computed as the midpoint of $\overline{AN}$ and $\overline{TB_M}$. Figure 9 shows how the triangle-splitting method executes at triangle t_r in Fig. 7.

3. Further Optimization

The method proposed in Sect. 2 takes long time because of unnecessary computations and vertex creation. In this section, we present optimization of our method to enable speedup.

As shown in Fig. 10, during the vertex-splitting step, we split each vertex into four vertices. Before splitting a vertex, we calculate a boundary sphere P for each child vertex. A boundary sphere outside the view frustum represents a vertex that will never be used again. Therefore, we only have to create vertices whose P's are not located outside the view frustum. Using this method, we can save much computation time via decreased the number of vertices.

Vertex splitting and triangle splitting involve many redundant computations. To reduce this redundancy, we use the z,w field of the vertex in these computations as a buffer, which is not used in the original method. We call this a *temporary buffer*.

Most of the computations in vertex splitting are performed in calculating the child vertex's coordinates, LOD selection, and VFC steps. If we have the block size for each vertex, and the LOD flag f, we can reduce the number of unnecessary computations.

We store the diagonal length of a block in the z field of its corresponding vertex. It can be used as the diameter of the bounding sphere. Since, we use square blocks in the terrain mesh, we can compute the location of the child vertex and perform VFC easily.

The w field stores the flag value f (see Sect. 2.1). During the vertex-splitting step, the same LOD selection process will be repeatedly executed for a vertex for which f is zero before $l = d$. By checking the vertex's w value before LOD selection, we can reduce the computation.

In triangle splitting, we have to compute the neighbor vertex and perform LOD selection in every splitting step. To simplify these computations, we also use the temporary buffers.

In Fig. 8's $\triangle ABC$ for example, the z field of vertex C stores the distance between N and T. Vertex A stores the predefined direction code (00: Up, 01: Down, 10: Left, and 11: Right) in the z field. This will simplify the computation of the neighbor vertex, by multiplying C's z value by 0.5.

While we repeatedly split the triangle, we have to check every neighbor vertex and its LOD selection, even if the triangle's splitting computation is already finished. By storing the flag f in the w value for vertex C, we can reduce redundant computation.

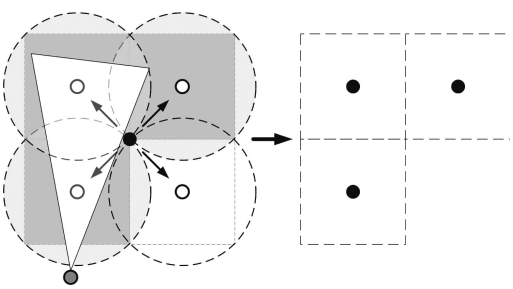


Fig. 10 Efficient vertex splitting using VFC. The fourth vertex located in the bottom right subblock is not created because its boundary sphere P is outside the view frustum.

4. Experimental Results

All tests were performed on a consumer PC equipped with an Intel Core 2 Duo E8400 CPU, 4 GB main memory, and an NVIDIA 9800 GTX (+) graphic card with 1 GB video memory.

The quadtree texture generation is performed by the CPU with an application programmed in C#. This process generates the quadtree texture as a 32-bit bitmap file, which is used for the main process. We implemented our rendering process in C++ and DirectX10 using a 32-bit operating system. The viewport size is 1024×768. Puget Sound (4097×4097) is used as the main dataset. Our rendering process is using in-core method to show that we can handle the large-sized quadtree in GPU. We can also render the larger terrain using out-of-core method by using the slices of the hight field with smaller quadtrees.

Table 1 shows the performance of our method. As the rendering speed is dependent on the viewing condition, we measured the rendering speed under two different viewing conditions, named **Far** and **Near**. We also measured rendering speed while changing the viewing directions. By using the Puget Sound dataset, the depth of the quadtree $d = 12$. We set the threshold τ for the projected error at one pixel and a half pixel. A-half pixel for τ represents the case of no pixel error (see also Eq. (2)). In the **Near** view, view point is located at the edge of the terrain in order to check the worst-case performance of our method. Figure 11 shows images of these two viewing conditions at $\tau = 1$. Thanks

Table 1 Changes of rendering speed according to viewing conditions (*fps*).

viewing angle	Near		Far	
	$\tau = 0.5$	$\tau = 1.0$	$\tau = 0.5$	$\tau = 1.0$
0°	8	19	8	24
30°	39	51	13	27
45°	60	74	17	39
60°	89	104	22	45
90°	170	243	32	72

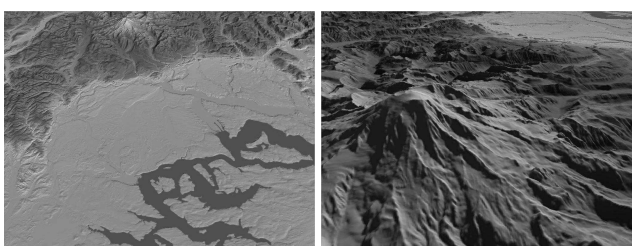


Fig. 11 Left image shows the result of **Far** view, and right image shows that of **Near** view.

Table 2 Performance enhancement when applying optimization.

τ	with optimization		w/o optimization	
	fps	# of triangle	fps	# of triangle
1.0	51	152 K	0	N/A
3.0	148	67 K	1	1718 K
10.0	501	16 K	3	1143 K

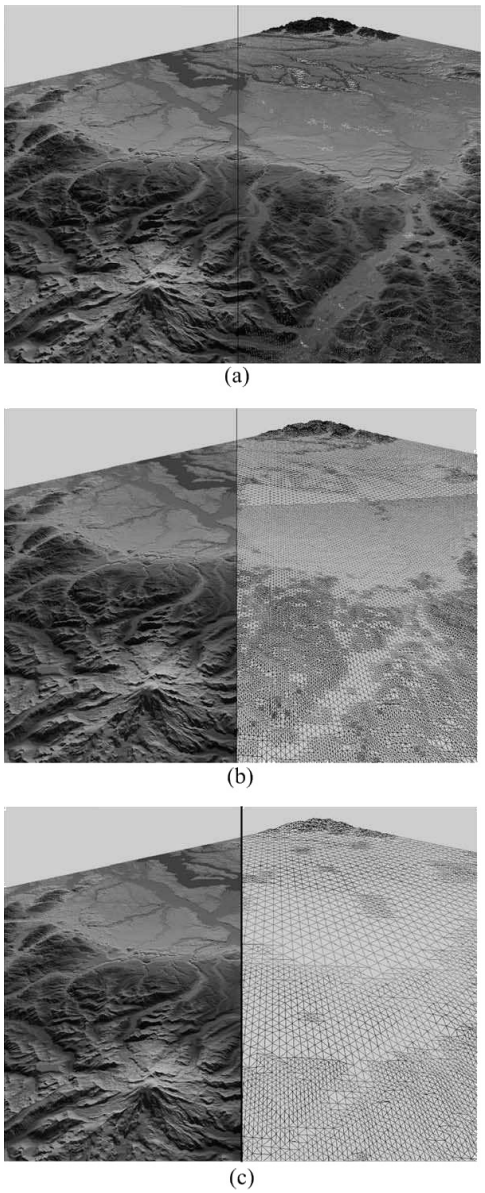


Fig. 12 Result images of the different thresholds. (a) τ = 1.0 (b) τ = 3.0 (c) τ = 10.0.

to VFC, in the **Near** view, it renders less triangles in comparison with **Far** view. As the viewing angles increase, the region of terrain included in the view frustum may decrease. Therefore rendering speed (*fps*) stiffly increases. The screen space error may increase when viewing angle gets lower and distance to terrain becomes shorter as shown in Table 1.

For acceleration, several optimization techniques described in Sect. 3 were applied. Table 2 shows the result of optimized method when the viewing angle is 30 degree and the viewing point is close the surface. Also it shows that frame rate increases when we apply bigger thresholds at the same viewing condition. Figure 12 shows images according to threshold values.

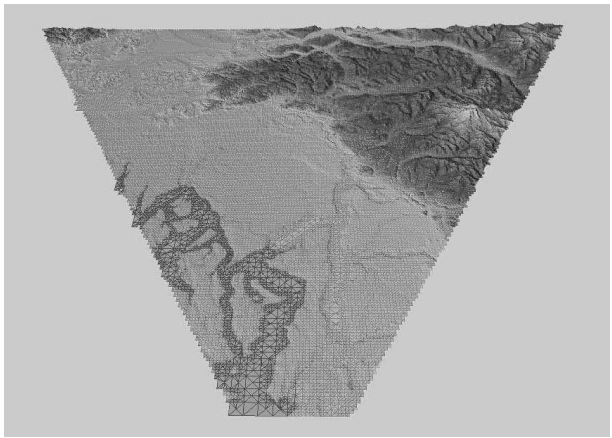


Fig. 13 Result of view frustum culling.

Table 3 Performance of our method when using various datasets.

Dataset	*d*	fps	# of Triangles
Puget Sound (4 K×4 K)	12	51	152 K
Grand Canyon (4 K×4 K)	12	29	231 K
Grand Canyon (2 K×2 K)	11	54	139 K
Je-ju Island (2 K×2 K)	11	114	83 K
Je-ju Island (1 K×1 K)	10	132	70 K

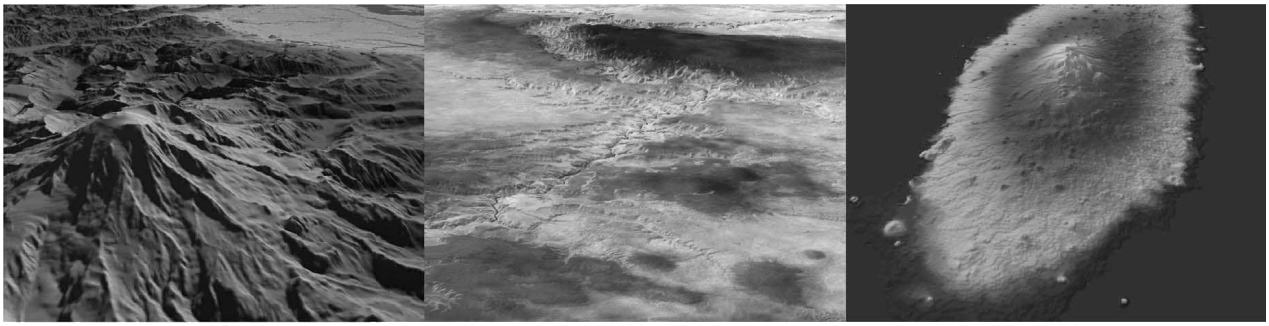
Figure 13 shows the wireframe representations of a terrain view applying VFC as the optimization technique. Because it removes the triangles out of the view frustum, rendering speed dramatically decreases.

Table 3 shows the performance evaluation using various datasets. Grand Canyon data is mountainous, and the Je-ju Island data (The biggest island of korea) is flat. The depth of the quadtree is in proportion to the size of the datasets. Experimental result shows that our method can render the flat dataset much faster. This is because flat terrain can be simplified more than the mountainous area. Figure 14 shows the result of various datasets applying our method.

Table 4 shows the CPU usage of our algorithm while changing viewing conditions with Puget Sound dataset. Since only the viewing condition is computed in CPU, CPU usage slightly increases only when the frame rate is very high.

Without triangle splitting, there might be some cracks as shown in Fig. 15 (a). Figure 15 (b) shows that all the cracks are removed when applying our triangle splitting.

When we apply the CPU-based quadtree triangulation [3] to our rendering method, it shows under 10 *fps* for Je-ju island dataset (resolution of data is 1025 × 1025, the τ = 1 and view point is far from the surface). Using another datasets, it takes over 1 second to render an image. This means that our GPU-based approach is at least 10 times faster than the CPU-based quadtree method.



(a) (b) (c)

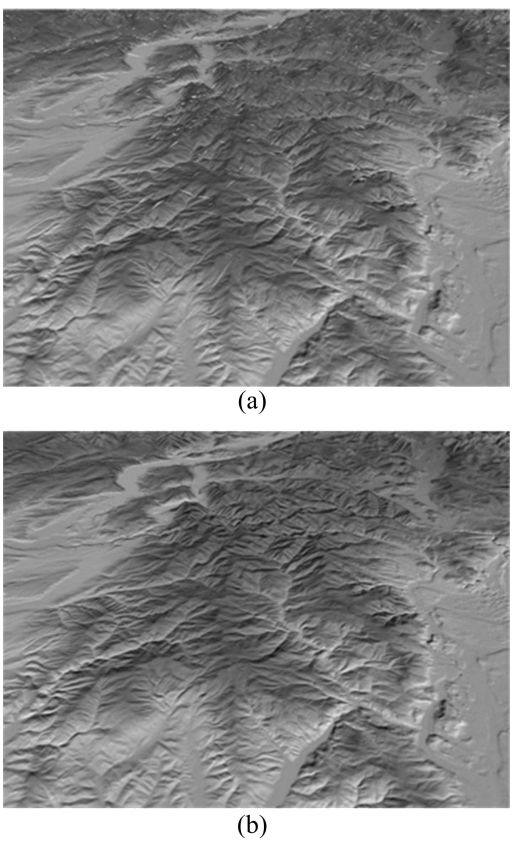
Fig. 14 Result images of the various datasets. (a) Puget Sound (4 K × 4 K). (b) Grand Canyon (2 K × 2 K). (c) Je-ju Island (1 K × 1 K).

Table 4 CPU usage of our method.

fps	CPU usage
19	0%
51	0%
74	0%
104	1%
243	2%

Table 5 A comparison of the number of triangles that has the same pixel errors.

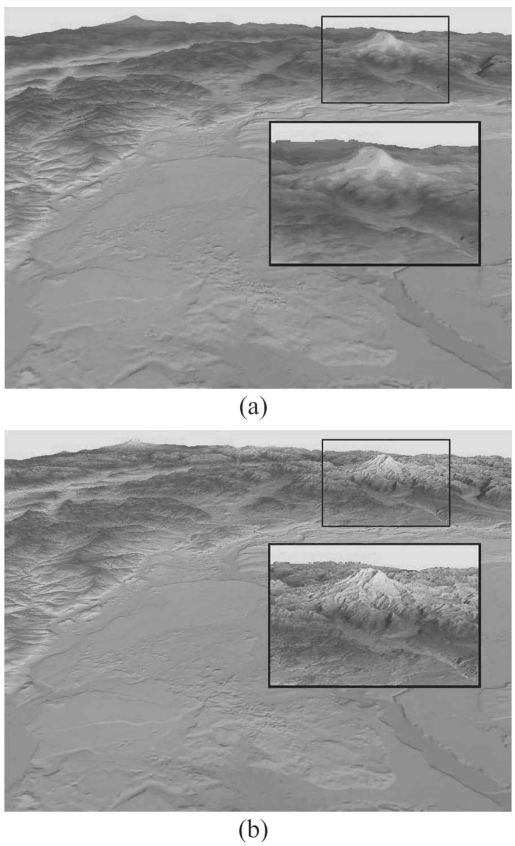
τ	PGM	Our approach
0	1280 K	627 K
1	540 K	152 K
2	720 K	105 K
3	320 K	67 K



(a)

(b)

Fig. 15 A comparison of images with (b) and without (a) triangle splitting. Our method can remove cracks efficiently.



(a)

(b)

Fig. 16 Comparison of geometry clipmap's result (a) and our result (b).

The previous methods such as geometry clipmap [29] and Persitant Grid Mapping (PGM) [32] show high rendering speed. However, they consider only the distance to surface, they will cause the geo-poppings due to varying screen space errors. To elliviate these problems, they require lots of triangles to render a scene. Our approach requires fixed screen space error, and shows less geo-poppings when using small threshold value.

Table 5 shows the maximum screen space error of PGM and triangles to be rendered comparing with our approach. It shows our approach can render a scene with less triangles. Therefore our method can reduce the video memory consuption than PGM under the same viewing conditions. We implemented PGM method with C# using XNA3.0 and we used the Puget Sound ($4\,\mathrm{K} \times 4\,\mathrm{K}$) dataset. We can measure the screen error of PGM by projecting an original vertex and its corresponding sampled vertex onto screen and measuring distance of the projected vertices [32]. We measured number of triangles of our method and PGM with the same viewing condition (viewpoint is near the surface and viewing angle is 30 degree). The maximum screen space error is referenced by the maximum pixel error of Puget Sound dateset in 1024×768 viewport [32].

We implemented geometry clipmap method in C# using XNA3.0 and we used the Puget Sound ($4\,\mathrm{K} \times 4\,\mathrm{K}$) dataset. Figure 16 (a) shows the result image of geometry clipmap and (b) for our method using the same dataset. By using the distance-based LOD selection, the clipmap-based method produces blurred images as shown in Fig. 16 (a). However, since our rendering method considers the mipmap level using the LOD selection which is used in the vertex splitting stage, our method shows the higher quality images in mountainous surface than those of clipmap-based methods [28]–[31].

5. Conclusions and Future Work

We have presented an efficient quadtree-based terrain-rendering method that provides higher quality and faster mesh generation than previous approaches. GPU-based geometry splitting is used to replace the quadtree. This enables parallel computation for the nodes and achieves higher performance than previous methods. By using our method, we can more greatly reduce transmisson of the vertices from CPU to GPU than the conventional. It can secure times for uploading the large-sized terrain datasets, and it can also save the video memory by simplification. In future work, we will investigate the single-pass algorithm for higher performance with frame coherency in various viewing conditions.

Acknowledgment

This work was supported by INHA university research grant.

References

[1] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner, "Real-time continuous level-of-detail ren-dering of height fields," Proc. ACM SIGGRAPH 1996, Addison Wesley, 1996.

[2] R. Lario, R. Pajarola, and F. Tirado, "HyperBlock-QuadTIN: Hyper-block quadtree based triangulated irregular networks," Proc. IASTED VIIP, pp.733–738, 2003.

[3] S. Rottger, W. Heidrich, P. Slusallek, and H. Seidel, "Real-time generation of continuous levels of detail for height fields," Proc. 6th International Conference in Central Europe on Computer Graphics and Visualization, 1998.

[4] R. Pajarola, "Large scale terrain visualization using the restricted quadtree triangulation," Proc. IEEE Visualization 1998, pp.19–26, 1998.

[5] R. Pajarola, M. Antonijuan, and R. Lario, "QuadTIN: Quadtree based triangulated irregular networks," Proc. IASTED VIIP, pp.733–738, 2003.

[6] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Al-drich, and M. Mineev-Weinstein, "ROAMing terrain: Real-time optimally adapting meshes," Proc. Visualization 1997, pp.81–88, 1997.

[7] A. Pomeranz, ROAM using triangle clusters, Dissertation, University of California at Davis, 2000.

[8] M. White, "Real-time optimally adapting meshes: Terrain visualization in games," International Journal of Computer Games Technology Volume 2008, Article ID 753584, 2008.

[9] P. Lindstrom and V. Pascucci, "Terrain simplification simplified: A general framework for view-dependent out-of-core visualization," IEEE Trans. Vis. Comput. Graphics, vol.8, no.3, pp.239–254, 2002.

[10] S. Basu and J. Snoeyink, "Terrain representation using right-triangulated irregular networks," 19th Canadian Conference on Computational Geometry, 2007.

[11] W.S. Evans, D.G. Kirkpatrick, and G. Townsend, "Right-triangulated irregular networks," Algorithmica, vol.30, no.2, pp.264–286, 2001.

[12] L.M. Hwa, M.A. Duchaineau, and K.I. Joy, "Adaptive 4-8 tex-ture hierarchies," Proc. Visualization 2004, pp.219–226, 2004.

[13] B. Purnomo, J.D. Cohen, and S. Kumar, "Seamless texture atlases," Symposium on Geometry Processing, pp.65–74, 2004.

[14] C.C. Tanner, C.J. Migdal, and M.T. Jones, "The clipmap: A virtual mipmap," Proc. ACM SIGGRAPH 1998, pp.151–158, 1998.

[15] J. Dollner, K. Baumann, and K. Hinrichs, "Texturing techniques for terrain visualization," Proc. Visualization 2000, pp.227–234, 2000.

[16] H. Hoppe, "Smooth view-dependent level-of-detail control and its application to terrain rendering," Proc. Visualization 1998, pp.35–42, 1998.

[17] R. Lario, R. Pajarola, and F. Tirado, "HyperBlock-QuadTIN: Hyper-block quadtree based triangulated irregular networks," Proc. IASTED VIIP, pp.733–738, 2003.

[18] J. Levenberg, "Fast view-dependent level-of-detail rendering using cached geometry," Proc. Visualization 2002, pp.259–266, 2002.

[19] T. Ulrich, "Rendering massive terrains using chunked level of detail control," Proc. ACM SIGGRAPH 2002, 2002.

[20] E. Yusov and V. Turlapov, "GPU-optimized efficient quad-tree based progressive multiresolution model for interactive large scale terrain rendering," Proc. GraphiCon2007, pp.53–60, Moscow, June 2007.

[21] C. Dick, J. Schneider, and R. Westermann, "Efficient geometry compression for GPU-based decoding in realtime terrain rendering," Computer Graphics Forum, vol.28, no.1, pp.67–83, 2009.

[22] Y. Li and J.H. Gong, "Global terrain data organization and compression methods," International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, vol.37, no.B5, pp.659–669, 2008.

[23] S. Kim, Y. Kim, M. Cho, and H. Cho, "A geometric compression algorithm for massive terrain data using delaunay triangulation," Proc. WSCG 1999, pp.124–131, 1999.

[24] C. Dick, J. Schneider, and R. Westermann, "Efficient geometry compression for GPU-based decoding in realtime terrain rendering," Computer Graphics Forum, vol.28, no.1, pp.67–83, 2009.

[25] S. Patidar, S. Bhattacharjee, J.M. Singh, and P.J. Narayanan, "Exploiting the Shader model 4.0 architecture," Technical Report IIIT Hyderabad, 2006.

[26] J. Schneider and R. Westermann, "GPU-friendly high-quality terrain rendering," J. WSCG, pp.49–56, Plzen-Bory Czech Republic, 2006.

[27] Y. Livny, Z. Kogan, and J. El-Sana, "Seamless patches for GPU-based terrain rendering," Proc. WSCG 2007, pp.201–208, 2007.

[28] F. Losasso and H. Hoppe, "Geometry clipmaps: Terrain render-ing using nested regular grids," ACM Trans. Graphics., vol.23, no.3, pp.769–776, 2004.

[29] A. Asirvatham and H. Hoppe, "Terrain rendering using GPU-based geometry clipmaps," GPU Gems 2, pp.27–45, 2005.
[30] M. Clasen and H. Hege, "Terrain rendering using spherical clipmaps," Eurographics/IEEE-VGTC Symposium on Visualization, 2006.
[31] S. Bhattacharjee and P.J. Narayanan, "Hexagonal geometry clipmaps for spherical terrain rendering," Proc. SIGGRAPH Asia 2008, 2008.
[32] Y. Livny, N. Sokolovsky, T. Grinshpoun, and J. El-Sana, "Persistent grid mapping: A GPU-based framework for in-teractive terrain rendering," Visual Computer, vol.24, pp.139–153, 2008.
[33] C.C. Tanner, C.J. Migdal, and M.T. Jones, "The clipmap: A virtual mipmap," Proc. ACM SIGGRAPH 1998, pp.151–158, 1998.
[34] H. Samet, "The quadtree and related hierarchical data struc-tures," ACM Comput. Surv., vol.16, no.2, pp.187–260, 1984.
[35] S. Patidar, S. Bhattacharjee, J.M. Singh, and P.J. Narayanan, "Exploiting the Shader model 4.0 architecture," Technical Report IIIT Hyderabad, 2006.
[36] B. Purnomo, J.D. Cohen, and S. Kumar, "Seamless texture atlases," Symposium on Geometry Processing, pp.65–74, 2004.
[37] X. Gu, S.J. Gortler, and H. Hoppe, "Geometry images," SIGGRAPH'02, pp.355–361, 2002.
[38] K. Niski, B. Purnomo, and J. Cohen, "Multi-grained level of detail using a hierarchical seamless texture atlas," ACM Symposium on Interactive 3D Graphics and Games, pp.153–160, 2007.
[39] H. Chang and B. Shin, "Hardware acceleration of terrain visualization using ef-Buffers," International Symposium on Computer and Information Sciences, vol.4263, pp.316–324, 2006.

Eun-Seok Lee recieved the B.S. degree in computer and information engineering from Inha University, Korea, in 2008. He is a M.S candidate in computer and information engineering at Inha University. His research interests include terrain visualization and hardware-based rendering.

Byeong-Seok Shin is an assistant professor in the school of computer and information engineering, Inha University, Korea. Current research interests include volume rendering, real-time graphics, and medical imaging. He received B.S., M.S., and Ph.D. in computer engineering from the Seoul National University in Korea.