Bimodal Vertex Splitting: Acceleration of Quadtree Triangulation for Terrain Rendering

SUMMARY Massive digital elevation models require a large number of geometric primitives that exceed the throughput of the existing graphics hardware. For the interactive visualization of these datasets, several adaptive reconstruction methods that reduce the number of primitives have been introduced over the decades. Quadtree triangulation, based on subdivision of the terrain into rectangular patches at different resolutions, is the most frequently used terrain reconstruction method. This usually accomplishes the triangulation using LOD (level-of-detail) selection and crack removal based on geometric errors. In this paper, we present bimodal vertex splitting, which performs LOD selection and crack removal concurrently on a GPU. The first mode splits each vertex for LOD selection and the second splits each vertex for crack removal. By performing these two operations concurrently on a GPU, we can efficiently accelerate the rendering speed by reducing the computation time and amount of transmission data in comparison with existing quadtree-based rendering methods.

key words: terrain rendering, level-of-detail, quadtree triangulation, real-time rendering

1. Introduction

Real-time visualization of massive terrain data is the most important issue in many applications, such as flight simulations, 3D games, and GIS (geographic information system). The massive dataset usually requires long processing time on CPU, so GPU-based techniques [1], [2] on this area have been continuously presented for speed up. However, due to the increasing resolution of the DEM (digital elevation model) and its ortho-photo texture data, more memory and computing power are required even when using the current high-end GPUs.

To overcome this problem, a number of out-of-core CLOD (continuous level-of-detail) methods such as geometry clipmap [3], [4] are continuously being investigated. These methods usually use a geometry cache to reduce the transmission data. The rendering speeds of these methods are faster than conventional triangulation methods [5]–[8], but they do not reduce the number of triangles efficiently. However, as the rendering area grows, the required number of triangles will increase, and may cause a serious bottleneck while rendering. In this paper, we solved this problem by reducing the number of triangles by using a parallel quadtree triangulation method using GPU.

Quadtree triangulation [5]–[9] is the most frequently used CLOD method for terrain rendering, supporting efficient reconstruction of an optimal mesh in real time. Conventional quadtree triangulation methods are usually performed on a CPU as the pointer and recursive operation are not suitable for a GPU. In our previous research, we proposed Geometry Splitting [7], which provides parallel quadtree triangulation using a GPU in order to take advantage of recent improvements in GPUs. The vertices are split in a geometry shader in order to select the detail level of the terrain mesh, and then they are converted into regular patches (vertex splitting). The triangles are then split to remove the cracks between patches that have different detail levels (triangle splitting). Since the entire process is performed on a GPU, this method provides high-speed rendering with an adaptively reconstructed terrain mesh.

However, a serious bottleneck occurs in geometry splitting at the stage of retransmission of stream output data. While splitting the geometric primitives, the data from the stream output stage have to be transmitted back to the input of the splitting process. As more transmissions occur, a greater number of splitting process repetitions occur. Geometry splitting performs vertex splitting and triangle splitting separately because cracks may appear after vertex splitting since they appear between adjacent patches with different detail levels. In addition, vertex and triangle primitives cannot be processed concurrently on a GPU. Therefore, triangle splitting has to be followed by vertex splitting. Thus, the triangle splitting process has to repeat the splitting process as many times as the vertex splitting. According to this repetition, transmission data and computing time will be increased. Geometry splitting is not appropriate for the out-of-core environment because of this drawback.

To alleviate this problem, we present bimodal vertex splitting, which considerably reduces the amount of computation and data transmission by performing LOD (level-of-detail) selection and crack removal concurrently on a GPU. To achieve this, we homogenize the type of primitives by using vertices with two different types. For LOD selection, we use L-type vertex (LOD selection-type vertex) and for crack removal we use C-type vertex (Crack removal-type vertex). In each splitting step, we check whether the corresponding terrain patch of an L-type vertex has already reached an adequate LOD. If not, we split the L-type vertex to search for its adequate LOD; otherwise, we have to convert it into the four C-type vertices that constitute the triangles of the terrain patch, and split them when the crack appears.
Our method efficiently reduces data transmission and computation time. In addition, the data could be reduced further due to compression of the triangle. According to our experimental results, we improved the frame rate more than six-fold on average as compared to state-of-the-art terrain rendering methods.

In Sect. 2, we introduce related work. The main algorithm is described in detail in Sect. 3. In Sect. 4, the experimental results are given, and finally, we present the conclusions of our study.

2. Related Work

A TIN (triangle irregular network) is a data structure for representing the terrain surface using irregular triangles. TIN is often used for traditional CPU-based triangulation method [10]–[16] that can achieve the optimal approximation of a terrain with the minimum number of triangles for the required accuracy. Hyper-triangulation [11], QuadTIN [12], and vertex hierarchy [13] are widely known hierarchies for TIN. Even though TIN tends to be more optimal than regular grid hierarchies, TIN requires much heavier preprocessing for triangulation and crack removal.

Regular grid hierarchy is a famous data structure for traditional CPU-based CLOD techniques, using uniform rectangular patches to represent terrain data. CLOD is the mesh simplification method which continuously evaluates according to a tradeoff between visual quality and performance. Compared with the TIN-based methods [10]–[16], it supports faster preprocessing and mesh reconstruction. Additionally, the data structures are based on a regular grid such as HRT (hierarchies of right triangulations) [16], longest edge bisection [5], [6], [16], and restricted quadtree triangulation [8], which efficiently simplify the memory layout and accelerate the rendering speed with top-down or bottom-up traversal.

In recent years, the performance of GPUs has been continually improving. As compared with a CPU, a GPU is more geared toward fast graphic operations and parallel computations. Therefore, in GPU-based methods, the main issue is how to reduce the transmissions between the main memory and the video memory. Like to regular grid techniques, there are patch-based CLOD techniques [17]–[20]. These methods improved the communication speed between a CPU and a GPU by using a geometry cache, which is the predefined geometry of a required region. Chunk LOD [17], [18] and BDAM [14] are famous techniques that efficiently improve the rendering speed and image quality using a geometry cache.

Advances in the programmability of GPUs have led to various GPU-friendly techniques. Geometry clipmap algorithms [3] render the surface by using tiles with various resolutions. The tiles are modified according to the updated clipmap data in each frame. Clipmap [21] is dynamic texture representation technique for real-time rendering. It caches the same sized clipped region of each level of mipmap to represent arbitrarily large size terrain in a finite amount of physical memory. To save more memory space, 4-8 texture [22] is introduced.

Since the size of graphic memory is limited, these methods involve communication overheads between the CPU and GPU. To alleviate this problem, Schneider et al. proposed the progressive transmission of geometry, which decreases the communication time between a CPU and a GPU [23]. Livny et al. suggested the use of seamless patches to reduce the communication time [20]. The predefined patches are stored in a cache and stitched between triangular tiles each of which has different LOD levels. Dick et al. proposed an efficient real-time GPU-based decoding method [17] that compresses the geometric data using a CPU, and decodes them in real time using a geometry shader. Thus, this method can utilize the graphic memory more efficiently and reduce the bottlenecks in data access and rendering.

By using the tessellator stage in shader model 5.0, we can refine the coarse input mesh to detail mesh on GPU [24], [25]. Crevin suggested the adaptive tessellation method [24] using the density map which stores the surface roughness of terrain patch. This method provides high speed mesh reconstruction that tessellates each patch adaptively by using density map on a GPU. However, previous sequential mesh reconstruction methods such as quadtree triangulation or TIN still provide a more optimal number of triangles.

On the other hand, a method using both a CPU and a GPU for LOD was proposed [26], performing the LOD selection through load balancing of a CPU and a GPU. The restricted quadtree triangulation [8] and the hierarchical seamless texture atlas [27] are used for LOD in CPUs and GPUs.

Due to the increasing size of display devices and terrain data, ray casting [28], which renders the terrain using the ray and height-field intersection instead of the polygonal mesh, has been proposed. This method provides a high-quality scene without using the geometric data. However, ray casting consumes too much time to compute the intersection of the ray and height field. To accelerate ray casting, quadtree-based empty space skipping [29]–[32] and hybrid approaches [33]–[35] have been proposed.

Quadtree-based empty space skipping exploits the fast GPU-based off-line processing by generating a quadtree as an image pyramid [29], [30]. The maximum height values of each node are stored in quadtree to check the empty space by using the bounding volume of the node. In each tree traversal, it checks the intersection between the ray and the bounding volume of each quadtree node. If there was no intersection, the bounding volume of the node will be the empty space. By skipping these empty spaces, this method efficiently accelerates the rendering speed.

Usually, in flat area of terrain, mesh-based rendering techniques show better performance than the ray casting methods. In contrast, ray casting supports better performance than the mesh-based approach in rough area. Hybrid methods [33]–[35] exploit this characteristic to improve the rendering speed, by applying mesh reconstruction on flat area, and ray casting on rough area. However, there is no
criterion for classification of terrain roughness since most of these methods are dependent of heuristics to classify the roughness of terrain.

3. Our Work

Usually, recursions and pointer operations of quadtree triangulation cannot be performed on GPU. Our previous work on geometry splitting [7] provides GPU-based quadtree triangulation overcomes this limitation. Figure 1 (a) shows the entire process of geometry splitting. First, we generate the quadtree texture on a CPU by using original DEM data at the pre-processing stage. Quadtree texture and DEM data will be used for the main texture resources of our process.

In the vertex splitting stage, we use a vertex as a pointer to traverse the quadtree in quadtree texture. This stage is operated on geometry shader and stream output buffer. Vertex primitives which store the address of specific node are used as the input type of geometry shader. To parallelize the tree traversal, we split each vertex into four vertices which point the child nodes in geometry shader and store the vertices at stream output buffer. We feedback these vertices as the input data of vertex splitting stage for the next detail levels. We perform the LOD selection by repeating this stage as many times as the depth of the quadtree.

We generate terrain mesh by converting vertices into quadrangle patches. However, the cracks may appear between adjacent patches with different detail levels. To remove those cracks, we split the triangle whenever its neighboring patch has a finer detail level. As the vertex splitting stage starts, the triangle splitting stage also needs to traverse the entire quadtree. Therefore, we have to traverse the tree twice in geometry splitting due to the heterogeneity of primitives, vertex primitives, and triangle primitives. The more times that this repeats, the more output geometry will cause performance degradation of GPU. Also a triangle consisted of three vertices, so the output data of triangle splitting stage is much bigger than at the vertex splitting stage.

To reduce the retransmission, we propose bimodal vertex splitting, which is shown in Fig. 1 (b). This method uses pyramidal quadtree texture which is quadtree texture which is stored as the form of image pyramid. So it is suitable for out-of-core processing which is based-on image pyramid such as clipmap [21] or 4-8 texture [22]. Since pyramidal quadtree texture can be generated on GPU by a bottom-up order, it can be generated faster than quadtree texture of geometry splitting. Also by removing the empty pixels of quadtree texture not used for tree traversal, we can effectively save the GPU memory consumption.

Bimodal vertex splitting requires only single tree traversal on a GPU for triangulation. To perform vertex splitting and triangle splitting concurrently, we unify the input primitive as vertices. We propose a technique to compress a triangle to a vertex. The bimodal vertex splitting stage checks for the adequate splitting mode of a vertex between vertex splitting (for LOD selection) and triangle splitting (for crack removal) in each process. This can efficiently reduce the number of tree traversal of geometry splitting into once.

After bimodal vertex splitting, we convert those vertices into terrain mesh, and render it directly to the screen. Unlike geometry splitting, this process does not need to output geometry to the main memory and we can save more transmission data.

3.1 Quadtree for GPU

As the entire process of bimodal vertex splitting is executed on a GPU, the quadtree should be stored in the GPU memory. Generally, a GPU cannot support the hierarchical data structures and serve the sufficient memory space, so CPU mainly does these actions. Therefore, in this section, we will introduce and describe the texture-based quadtree in detail and the simple error metric of bimodal vertex splitting.

In general, quadtree structures cannot be handled on a GPU, because GPUs do not execute the pointer operations.
We store and traverse the quadtree on a GPU memory by using the texture pyramid [3], [29] which is the most well-known hierarchy for GPU-based terrain rendering methods [3], [29]–[35].

A texture pyramid is the collection of multi-resolution images and is easily stored on GPU as a texture array. Each texel of the textures is regarded as a node of the quadtree. As shown in Fig. 2, the texture pyramid can substitute a quadtree, by linking a texel with the others of the adjacent texture level as the child or the parent nodes.

Our method adaptively reconstructs the terrain mesh by the viewing conditions which are changing frequently, meaning that we have to store the maximum geometric error of the terrain patch, which is shown in Fig. 3. The maximum geometric error value is used to compute the screen space error.

As shown in Fig. 3, there are five geometric error values, denoted as $\delta_{0-4}$, between two adjacent detail levels. Geometric error $\delta$ can be computed as the distance between the midpoint of an edge of the dashed grid (lower level) and the vertex of the lined grid (higher level), located at the same $xz$ position. The screen-space error can be calculated by measuring the distance of white vertices and black vertices on the screen space. Using the geometric error value, we can easily obtain the height of gray vertex by adding $\delta$ to the height of white vertex. Therefore, we should store the geometric error $\delta$ to determine whether a screen-space error occurs.

To estimate the maximum value of the screen-space error of a node, we also have to consider the geometric errors of its child nodes. When one of the child nodes has a geometric error that is bigger than that of its parent, we cannot calculate the accurate screen-space error, and it may cause some artifacts, such as the geo-popping effect. Since there are five geometric errors in each node, much computation will be required; storing all the geometric errors that may raise the memory consumption.

In geometry splitting, we used the representative error of a node $\delta_{rep}$ at the midpoint of the patch [7], $\delta_{rep}$ is chosen by the maximum value between the value of the geometric errors of the patch and double the value of the representative geometric error of the child nodes. Using double the geometric errors of the child nodes, the maximum error of the node can appear to be twice the size of a child node in the worst case. However, as $\delta_{rep}$ of a node stores double the value of its child nodes recursively, the maximum screen-space error will be measured as much larger than its actual error value that appears on the screen. Therefore, this error metric wastes excessive time on processing LOD selection due to inaccurate error computations.

To solve this problem, we use a different measuring technique that uses $\delta_{rep}$ of the current node and $\delta_{rep}$s of its children ($\delta_{0-4}$ in Fig. 3). We measure the screen-space error at a different position, noted $\delta_{rep}$, as depicted in Fig. 4. This position is closest to the bounding sphere of a patch, which is determined as the intersection point between the ray from view and the bounding sphere. Suppose that there are two objects of the same size and shape and we are projecting them into the screen space with the perspective projection. The projected area of the object at the closest position will be bigger than that of the other object at a different position. We can accordingly measure the accurate screen-space error $\delta_{rep}$ from $\delta_{rep}$, instead of using the double-sized geometric errors of the child nodes. In addition, this may reduce the excessive subdivisions generated from the inaccurate LOD selections that appear while using the accumulated double-sized geometric error at the lower detail levels.

Computation of $\delta_{rep}$ of a node requires the geometric errors of itself, and $\delta_{rep}$s of child nodes. Therefore, this computation is independent to the computation of other nodes in the same detail level. Since each computation of $\delta_{rep}$ has to regard the $\delta_{rep}$ of child nodes, we generate each level of pyramidal quadtree texture in bottom-up order on GPU. We store the rendering result of pixel shader (or fragment shader) as a texture. Since pixel shader computes the color value of every fragment in parallel, we efficiently generate a specific level of quadtree texture by mapping each node as a fragment. Firstly, we generate the lowest level
which contains the leaf nodes and upload it to GPU as texture. Then we generate the upper level texture using lower level texture which was generated at previous step. By repeating this, we can upload the quadtree on GPU as the pyramidal quadtree texture.

3.2 Bimodal Vertex Splitting

As seen in (b) of Fig. 1, bimodal vertex splitting concurrently performs the LOD selection and the crack removal process on a single tree traversal. The entire process of bimodal vertex splitting is performed on geometry shader of programmable rendering pipeline. To unify the input a primitive, we defined the vertex for LOD selection as L-type vertex, and vertex for crack removal in C-type vertex. In each splitting process, we select the appropriate mode by considering the type of a vertex on geometry shader. With this method, we can efficiently perform the LOD selection and crack removal at same step on geometry shader.

3.2.1 LOD Selection (L-type Vertex Splitting)

We defined the unified vertex layout with the position channel \((x, y, z)\) field and color channel \((r, g, b)\) field. To refer to the maximum geometric error of a node, the L-type vertex stores the \(u, v\) coordinates which points the center position of terrain patch at \(x\) and \(y\) field and detail level of the terrain patch at \(z\) field. In \(r\) field, we store the vertex type to figure out whether the vertex is L-type or C-type. \(g\) and whether the \(b\) field is only used for a C-type vertex. We will introduce the C-type vertex in the next section.

Usually in GPU-based top-down traversal of quadtree [30], [31], every thread traverses the quadtree from root node. This may cause duplicated traversal for several nodes. Our method solved this problem by traversing the quadtree using a vertex primitive as a pointer. By splitting a vertex on geometry shader as in Fig. 5, we can efficiently parallelize the quadtree traversal for detail level selection. Since the bimodal vertex splitting is a top-down traversal, a single L-type vertex becomes the initial input data of geometry shader. This vertex represents the root node of the quadtree. It will locate at the center of the terrain patch as the black vertex in Fig. 5. We refer to the maximum geometric error of pyramidal quadtree texture by using data stored in the \(x, y, z\) fields of L-type vertex. While traversing the tree, we split the L-type vertex into four L-type vertices as the second step of Fig. 5. Each of them will be located at the center of the corresponding sub-patches of the root node. After splitting the vertex, if we find L-type vertices where their corresponding patches are outside of the viewing frustum, we delete them to reduce the computation. The remaining vertices will be output to the main memory by the stream output stage, and they will be re-transmitted as an input data of the bimodal vertex splitting stage. By repeating these steps recursively, we can efficiently find the adequate detail level in parallel. When an L-type vertex reaches its adequate detail level, we convert it into C-type vertices to remove cracks while other remaining L-type vertices are finding their adequate detail levels.

3.2.2 Crack Removal (C-type Vertex Splitting)

While we perform the detail level selection on geometry shader, we cannot know the detail level of adjacent patches. Therefore, as depicted in Fig. 6, cracks appear when the adjacent patches have different detail levels, which is typical.

The gray vertex in Fig. 6 shows how a crack occurs. By selecting a detail level in top-down order, a crack may appear when a vertex stops the LOD selection while its neighbors should be subdivided further. To remove the crack, we have to reconstruct the patch with four isosceles right triangles as the lower level quadrangle in second step of Fig. 6. Then we split the triangle at the next step, causing a crack on its hypotenuse.

However, this technique has to be performed using triangle primitives in geometry splitting. Bimodal vertex splitting that uses C-type vertices solve this problem.

A T-vertex may appear at the side of a patch. To remove the cracks with triangle splitting, four triangles which
Fig. 7 Converting L-type vertex to C-type vertices. Black vertex is L-type vertex and gray vertices are the C-type vertices.

Fig. 8 Example of C-type vertex splitting of t_{right}. We eliminate cracks by splitting the C-type vertex while the neighboring patches subdivide by L-type vertex splitting. Dotted line represents the neighboring patch of C-type vertex.

Fig. 9 Properties of a C-type vertex that contains t_{right}.

\[ H' = H + \frac{h_{d-1}}{h_d} \]

3.3 Converting Vertices into Terrain Mesh

As a result of vertex splitting, a bunch of L-type vertices and C-type vertices will be generated. To convert these vertices into terrain mesh, we convert the L-type vertices to quadrangle patches, and the C-type vertices to triangles.

By using the detail level value of an L-type vertex, we can estimate the length of the side of terrain patches. In using the u, v coordinate, we can get the location of the patch.

In case of a C-type vertex, we convert it into a triangle by computing the position of three vertices which we stored in the C-type vertex. We compute one by using the u, v coordinate, and the other by using hypotenuse factor and detail level value. The last one can be computed by detail level of the adjacent patch. By considering the correlation of direction flags and the coordinate system of vertices of triangle, we can convert the rest of the triangles easily by rotation matrix.

This process was also performed on a geometry shader. In this pass, the amount of geometry data increases by converting the vertices into polygons. With our method, we directly render the output geometry using pixel shader instead of transmitting the geometry data to main memory by the stream output stage. This accelerates the total rendering.
speed by reducing retransmissions of geometry data.

4. Experimental Result

All the experiments were performed on a consumer PC equipped with Intel Core\textsuperscript{TM} i5-2500 3.3 GHz CPU, 8 GB of main memory, and nVidia\textsuperscript{TM} GeForce GTX 560 Ti graphic card with 1 GB of local graphic memory. We used the DirectX 10 and the shader model 4.0 as the graphics API. We used the 16-bit Puget Sound, Grand Canyon, and Jeju-island dataset. The viewport size was set to $1280 \times 1024$ pixels. Figure 10 is the resulting image of bimodal vertex splitting method.

Rendering the original data without any acceleration technique cannot render a scene in real-time. So, to prove the efficiency of our method, we tested several comparative experiments between our method and state-of-art methods. Since the limitation of the texture size of graphic hardware is $8192^2$ and the size of the dataset has to be $(2^d + 1)^2$, we limited the maximum resolutions of datasets as $4097^2$ which generates the biggest quadtree. The depth of the quadtree $d$ will therefore be 12 at the maximum size. First, we compared the rendering speed and the number of triangles in our method and the quadtree-based methods which is performed on on-the-board GPU environment. We compared the rendering speed of bimodal vertex splitting (BVS in Fig. 11), geometry splitting [7] (GS in Fig. 11), chunk level-of-detail method using ef-buffer [19] (Chunk in Fig. 11), ray casting using maximum mipmap [30], [31]; and hybrid ray casting using vertex propagation (Ray-VP in Fig. 11) [32].

Ray casting-based methods are not available for comparing the triangle number because they do not use mesh for rendering a terrain dataset. Therefore, we only compared the number of triangles of mesh-based method: bimodal vertex splitting, geometry splitting, and chunk LOD.

We set all the error tolerances at a half pixel to generate errorless scenes. For all scenes, we applied the various recorded flight viewing conditions to compare the rendering speed at the same viewing condition of each method. For the datasets, we use Puget Sound ($4097 \times 4097$), Grand Canyon ($2049 \times 2049$), and Jeju-island ($1025 \times 1025$) for this experiment.

As shown in Fig. 11, the x-axis of each graph (scene number) means the recorded viewing conditions which are stored while we move the flight overviewing camera. Bimodal vertex splitting makes an improvement in the rendering speed by approximately 343.2\% of geometry splitting, 523.5\% of ray-casting using maximum mipmap, 269.4\% of hybrid ray-casting using vertex propagation, and 459.3\% of chunk LOD on average. The number of triangles of bimodal vertex splitting produces 32.2\% less than geometry splitting and on average 51.1\% fewer triangles than chunk LOD on average. Chunk LOD usually uses chunk as a geometry cache, so it simplifies the given area and selects the detail level of chunk by means of the maximum error value of the chunk. Therefore, the conventional quadtree traversal produces lesser number of triangles as the results.

This implies that our method effectively improved the rendering speed and the triangle number of reconstructed terrain mesh without any screen space errors. By reducing the data transmission with single tree traversal, we can achieve much faster rendering speed than other methods. Also LOD selection of bimodal vertex traversal produces less number of triangles than geometry splitting.

To measure the improvement of a single tree traver-
Fig. 11 Performance comparison of bimodal vertex splitting and previous methods in the same recorded viewing condition.

Table 1 Comparison of the Maximum and Minimum fps of Geometry Splitting (GS) and Bimodal Vertex Splitting (BVS) Using the Same LOD Selection by Pyramidal Vertex Splitting.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>GS Max</th>
<th>GS Min</th>
<th>BVS Max</th>
<th>BVS Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Puget Sound (4097×4097)</td>
<td>127</td>
<td>53</td>
<td>226</td>
<td>122</td>
</tr>
<tr>
<td>Grand Canyon (2049×2049)</td>
<td>192</td>
<td>78</td>
<td>272</td>
<td>147</td>
</tr>
<tr>
<td>Jeju island (1025×1025)</td>
<td>253</td>
<td>133</td>
<td>282</td>
<td>172</td>
</tr>
</tbody>
</table>

Fig. 12 Number of transmitted vertices of (a) geometry splitting and (b) bimodal vertex splitting. The vertical axis shows the processing time (ms), and the horizontal axis, the step numbers.

Fig. 13 Number of transmitted vertices of (a) geometry splitting and (b) bimodal vertex splitting. The vertical axis shows the processing time (ms), and the horizontal axis, the step numbers.

In a sal, we first compared the rendering speed of bimodal vertex splitting and geometry splitting as Table 1. For a fair comparison, we used pyramidal quadtree texture and the error metric of bimodal vertex splitting for LOD selection of both two methods so the triangle number and screen space error will be exactly same. As seen in Table 1, the bimodal vertex splitting improved the rendering speed to 161% of geometry splitting on average. The improvement of minimum rendering speed is bigger than the maximum rendering speed. This means that we efficiently reduce the computational cost and retransmission data by single tree traversal.

Figure 12 shows the data transmission times of the stream output stage of geometry splitting and bimodal vertex splitting. The result images of these two methods are exactly same as shown in (a) of Fig. 10. We used the error metric of bimodal vertex splitting for both methods to compare the transmission time impartially. We used nVidia™ PerfHud v6.0 to estimate the processing time of each step. In Fig. 13, the transmission time of each rendering pass of geometry splitting and bimodal vertex splitting are shown. Bimodal vertex splitting requires less than half the draw calls as compared to geometry splitting, since it performs the LOD selection and the crack removal concurrently. In geometry splitting, pass 1-11 are the vertex splitting, pass 12 is the mesh conversion, pass 13-23 are triangle splitting, and pass 24 is the rendering step (see Fig. 12 (a)). In bimodal vertex splitting, pass 1-11 are the bimodal vertex splitting and pass 12 is mesh conversion and rendering. We shorten the transmission time by almost 81.92% by reducing the recursive processes and reducing the transmission data by compressing the triangles as C-type vertices.

An on-the-board GPU environment is not suitable for rendering massive terrain data which exceeds the GPU memory. To render these dataset, we have to render using out-of-core methods such as geometry clipmap and Persistent Grid Mapping (we call this PGM hereafter). We compared the rendering speed and the number of triangles of these methods and bimodal vertex splitting to prove the efficiency of our method in out-of-core environment. For impartial comparison, we applied the same sized clip region and clipmap level for these three methods. The clip region is 513 × 513, and the five detail levels are used for the clipmap. To apply the clipmap to bimodal vertex splitting, we used the input patch which covers the area of coarsest level of clipmap. Also we set a limit the number of splitting according to the number of clipmap levels. Also we have to stop the L-type vertices splitting and convert it to C-type vertices when the finer level of clipmap does not support more detail area.

The PGM refines the terrain mesh by projecting the regular grids from screen space to $xz$ plane. For a fair comparison, we have to define the vertex interval of initial regular grid to cover the resolution of clip region. So we set 2 pixels for vertex interval to cover $513 \times 513$ sized clip region in $1280 \times 1024$ viewport. So we can make a map every texels from clipmap using the refined mesh. This method uses fixed number of triangle (655,360 triangles).
Performance comparison of bimodal vertex splitting and previous methods in the same recorded flight overviewing condition. The vertical axis represents the fps and the horizontal axis depicts the scene number.

For the datasets, we use Puget Sound (16K×16K), Grand Canyon (8193×4096), and Jeju-island (8193×8193) for this experiment.

Figure 13 shows the result of comparison of bimodal vertex splitting and state-of-art out-of-core methods. As a result, we’ve improved the rendering speed over 132% of the geometry clipmap and 127% of the PGM on average. Bimodal vertex splitting only uses 31% and 19% triangles of geometry clipmap and PGM. Despite using the pyramidial quadtree texture, our method not only provides a faster rendering speed, but reduces the amount of geometry data than the conventional out-of-core methods.

5. Conclusion

In this paper, we suggest an efficient GPU-based quadtree triangulation method that performs LOD selection and crack removal concurrently, reducing the data to be transmitted and resolving the bottleneck of geometry splitting. In addition, compressing a triangle to a vertex could handle the LOD selection and crack removal in the same pass of the geometry shader. By performing LOD selection and crack removal simultaneously, we can dramatically lessen the processing time. Further, we can efficiently reduce the transmission data by using the compressed expression of the triangle information.

However, there are still much data transmissions between GPU memory and main memory, because of the recursion process. If we could render without the recursion, this would save much more processing time. In the future work, we improve our method by reusing the geometry data from previous frame to overcome this drawback.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No.45256-01).

References


Eun-Seok Lee received B.S. and M.S. degree in Computer Science from the Inha University of Korea in 2008 and 2010, Inha University, Korea. He is currently pursuing a Ph.D in Computer Science under the advisory of Prof. Byeong-Seok Shin. His research interest includes real-time graphics, terrain rendering and level-of-detail.

Jin-Hee Lee is a Ph.D candidate in Computer and Information Engineering, Inha University, Korea. She received the BS degree in Computer Science from Korea National Open University, and MS degree in Computer and Information Engineering from Inha University of Korea. Her research interests include human computer interaction, wearable computer, and computer graphics.

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.