LETTER

Geometry Clipmaps Terrain Rendering Using Hardware Tessellation*

Ge SONG[†a)], ***Member***, **Hongyu YANG**[†], ***and*** **Yulong JI**[††], ***Nonmembers***

SUMMARY Due to heavy rendering load and unstable frame rate when rendering large terrain, this paper proposes a geometry clipmaps based algorithm. Triangle meshes are generated by few tessellation control points in GPU tessellation shader. 'Cracks' caused by different resolution between adjacent levels are eliminated by modifying outer tessellation level factor of shared edges between levels. Experimental results show the algorithm is able to improve rendering efficiency and frame rate stability in terrain navigation.

key words: *terrain rendering, GPU, tessellation shader, geometry clipmaps*

1. Introduction

Real-time high quality rendering large scale terrain in stable frame rate is still a challenge problem in many applications such as flight simulation, virtual reality and so on. Classical methods with CPU can not be adapted to modern graphics hardware [1]. One of the common used GPU-based terrain rendering algorithm is called geometry clipmaps [2]. Its basic idea is to store terrain elevation data in form of image, which changes the issue of terrain grid dispatch into image reading. 3D structures of terrain is represented efficiently by view-dependent multi-resolution regular triangle meshes called pyramids. It avoids instability of frame rate caused by different local complexities terrain in details. With use of hight-performance GPU, [3] proposed an optimized method based on geometry clipmaps. Vertex buffer and index buffer are built in CPU and transferred into GPU, from which triangle meshes can be generated. However, due to large volume of information transferred from CPU to GPU, load unbalance between CPU and GPU is inevitable. Many other algorithms based on geometry clipmaps have been proposed, but none of them can solve the problem fundamentally [4]–[6].

[7] proposed a terrain rendering algorithm using hardware tessellation. In this algorithm, tessellation shader can generate triangles adaptively with only a few controlling vertices from CPU. However, this method suffers from the problem of sudden changing of frame rate during the process of high speed navigation because it generated meshes from local terrains with different geometry complexities.

This paper proposes a novel method to render terrain of ultra-high resolution in real-time with smooth frame rate. It takes advantage of both geometry clipmaps and hardware tessellation. In contrast to the previous methods, our representation of terrain is simple and efficient. Moreover, with use of hardware-supported tessellation, elimination of cracks is very easy to realize.

2. Algorithm Overview

For each level of geometry clipmaps, we create only a few tessellation control points which stored in vertex buffer to represent basic mesh framework.

Index buffer is filled with tessellation control points and adaptive triangle meshes are generated. In update stage, we need only replace a few indices in index buffer to change status of geometry clipmaps.

We eliminate cracks by increasing the value of outer tessellation level factor of the shared edges between levels in use of tessellation shader.

3. Algorithm Implementation

3.1 Creation of Tessellation Control Points

For each level in geometry clipmaps, a square ring is built (shown in Fig. 1), which is divided into several blocks for clipping of FOV. These blocks include 3 types: twelve m*m squares (grey part shown in Fig. 1, called M-block), four m*2 rectangles (green part called R-block) and one L-shape block (blue part called L-block). L-block may be positioned at left-up, left-bottom, right-up or right-bottom around M-Block.

Different from [2] and [3], our method needs only to create points as many as enough to represent basic structure of geometry clipmaps, instead of creating whole vertices of primitives. Tessellation control points will be transferred to GPU by being written into vertex buffer from CPU. Patches are generated by tessellation shader and compose every triangle.

The distribution of tessellation control points is shown in Fig. 1, where $\{n0, n1, \ldots, n31\}$ cover both M-blocks and R-blocks; $\{n32, n33, \ldots, n43\}$ are created to cover L-block

Manuscript received July 13, 2016.
Manuscript revised October 21, 2016.
Manuscript publicized November 9, 2016.

[†]The authors are with the National Key Lab. of Fundamental Sci. on Synthetic Vision, Sichuan University, Chengdu, China.
[††]The author is with the School of Aeronautics & Astronautics, Sichuan University, Chengdu, China.

*This work was done when Song was a doctoral candidate at Sichuan University, China.

a) E-mail: songge86@yeah.net

DOI: 10.1587/transinf.2016EDL8160

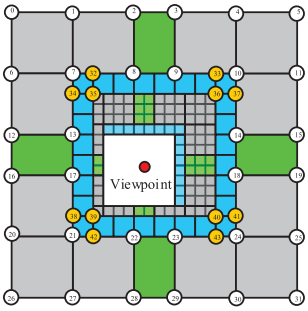

Fig. 1 Distribution of tessellation control points.

which may exist at 4 different position. After stored in vertex buffer, tessellation control points would be quoted by index buffer of each level of geometry clipmaps to generate triangle patches.

3.2 Generation and Update of Adaptive Triangle Patches

In geometry clipmaps, terrain data is treated as a 2D elevation image, which is prefiltering into a mipmap pyramid of L levels [2]. Without sampling height value for every vetex from mipmaps in vertex shader, we postpone it into tessellation evaluation shader stage until which patches have been generated. Therefore, there is no detail loss of sampling precision.

There is an index buffer in which indices of vertices are stored for each level of geometry clipmaps. Adaptive triangle patches are generated by these indices and updated in real time.

Three types of tessellation domain are supported by OpenGL: quads, triangles and contour sets. We use quads to generate patches because all blocks of geometry clipmaps are rectangles. Therefore, points in index buffers would be transferred in fours to tessellation control shader. M-blocks and R-blocks are built with 1 group of indices (four points), while L-blocks are built with 3 groups, which represent horizontal part, vertical part and the corner part in size of 1 respectively. L-blocks are going to be built in four ways (shown in Fig. 2) according to their different positions while real-time navigation.

Assume geometry clipmaps is built by an n*n matrix of dots, the size of M-blocks is:

$$m = (n+1)/4 - 1 \tag{1}$$

Accordingly, the length of long side of R-blocks is m and the length of short side is $n - 4m - 1 = 2$.

The width of L-blocks is 1, and the length of horizontal and vertical parts is $2m + 1$.

Thus tessellation levels of 3 types of blocks can be obtained:

$$\begin{cases} T_M^{inner}[i] = m, & \text{i} = 0, 1 \\ T_M^{outer}[i] = m, & \text{i} = 0, 1, 2, 3 \end{cases} \tag{2}$$

$$\begin{cases} T_R^{inner}[0] = T_R^{outer}[0] = T_R^{outer}[3] = 2 \\ T_R^{inner}[1] = T_R^{outer}[1] = T_R^{outer}[2] = m \end{cases} \tag{3}$$

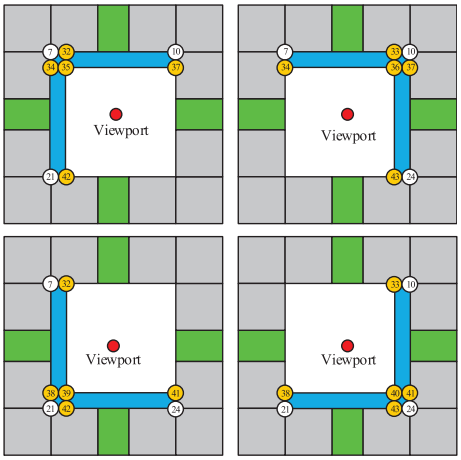

Fig. 2 4 types of structures of L-blocks.

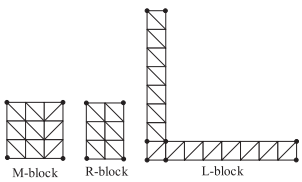

Fig. 3 Example of adaptive triangle patches.

$$\begin{cases} T_L^{inner}[0] = T_L^{outer}[0] = T_L^{outer}[3] = 1 \\ T_L^{inner}[1] = T_L^{outer}[1] = T_R^{outer}[2] = 2m + 1 \end{cases} \tag{4}$$

Assume $n = 15$, adaptive triangle patches is generated like what is shown in Fig. 3, in which the black dots represent tessellation control points.

From the foregoing discussion, M-blocks or R-blocks can be built within 4 control points and L-blocks can be built within 8 control points regardless of any value of n.

While real-time navigation, only L-block of each level needs to be updated every time because movement of viewpoint is continuous.

Further, the required step for update of each level increases exponentially from inner to outer. Assume the required step for update at innermost level (i.e., the finest level) is s, the required step for update of level l_k is:

$$s_k = s \times 2^k \tag{5}$$

Therefore, updates of adaptive triangle patches occur rarely. Even if an update occurs, only 8 control points of L-blocks need to be updated instead of replacing the whole vertices of L-blocks in classical method.

3.3 Elimination of Cracks between Adjacent Levels

Classical geometry clipmaps algorithm renders a string of degenerate triangles to avoid cracks. This method would not only cause distortion in visual effect, but also bring additional rendering burden. However, we can solve this problem easily and efficiently by increasing the value of outer tessellation level factor of the shared edges between levels

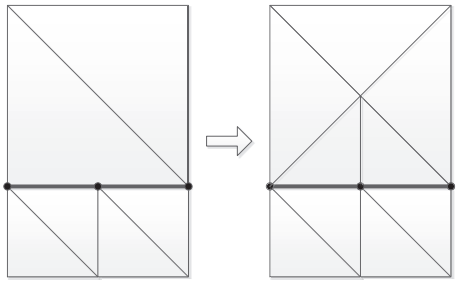

Fig. 4 Principle of cracks elimination.

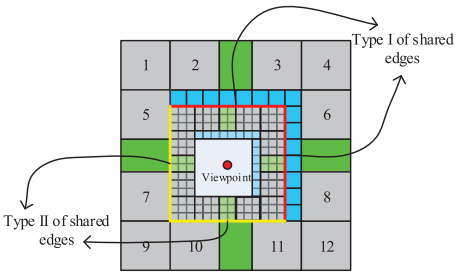

Fig. 5 Shared edges between adjacent levels.

in use of tessellation shader.

The inter tessellation level factor set in Eqs. (2) (3) (4) is consistent with the outer one, which means generated patches are average. However, the vertex number of inter level is twice that of out lever because of different resolutions of adjacent levels, which is the root cause of cracks. We solve this problem by enlarging the outer tessellation level factor on shared edges. Figure 4 gives the principle of cracks elimination.

There is 2 types of shared edges between adjacent levels: Type I is at L-block and Type II at M-block and R-block on the other side (as shown in Fig. 5).

For Type I, we need only set tessellation level factor for L-block as follow:

$$T_L^{outer*}[in] = T_L^{outer}[in] \times 2 \tag{6}$$

Where *in* is the shared edge of L-block.

For Type II, all blocks on shared edges are also operated in same way of Type I. However, there is a little part of M-block 5 and M-block 11's inner edges is on the side of L-block. New cracks show up if we did as above. For brevity, we update tessellation level factor of the corresponding edges of L-block by Eq. (6), that is to say, the value of tessellation level factor set to 2.

Figure 6 give a example when n = 15, where it shows the result of cracks elimination with the way of setting tessellation level factor of shared edges between adjacent levels. As shown, patches on shared edges are docked each other between levels and no cracks come out.

4. Experimental Results and Discussion

The algorithm proposed in this paper has been implemented

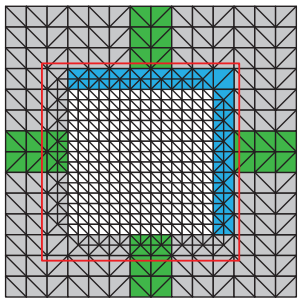

Fig. 6 Example of how to eliminate cracks.

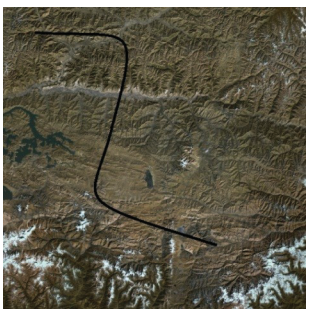

Fig. 7 Navagation path in experiments.

Table 1 Comparison of vertex number from CPU to GPU

LOD Levels = 7			LOD Levels = 9		
Size of n	Ours	Classical	Size of n	Ours	Classical
63*63	301	27783	63*63	387	35721
127*127	301	112903	127*127	387	145161
255*255	301	455175	255*255	387	585225
511*511	301	1827847	511*511	387	2350089

on a workstation with the following hardware configuration: Intel Core i7-2600K @3.4, 8.0GB RAM and NVidia GTX 590. To test our algorithm, the resolution of height maps and image maps is 16385*16385 and 16384*16384 respectively.

The first experiment is conducted to compare the number of vertices transferred from CPU to GPU between our algorithm and geometry clipmaps. From Table 1, we learn that vertex number is very large in classical geometry clipmaps. And it will increase in squared growth rate with the increase of size of n. However, vertex number in our algorithm is very small and irrelevant to the size of n because we need only to transfer several tessellation control points to GPU.

The second experiment is conducted to compare the frame rate and its stability among many algorithms including ours. The navagation path is as shown in Fig. 7. We learn from Fig. 8 that our frame rate was much higher than [3] and [4]. [7]'s average rendering performance was about same with ours, but its frame rate is quite unstable, especially with a significant decrease around 180th sec and 350th sec, which was corresponded to twice turn in roaming path: sudden changing of view direction lead to split-second variation of LOD in FOV, which caused linear reduction of rendering performance. However, our algorithm inherits geometry clipmaps' strong points of stable frame rate, while

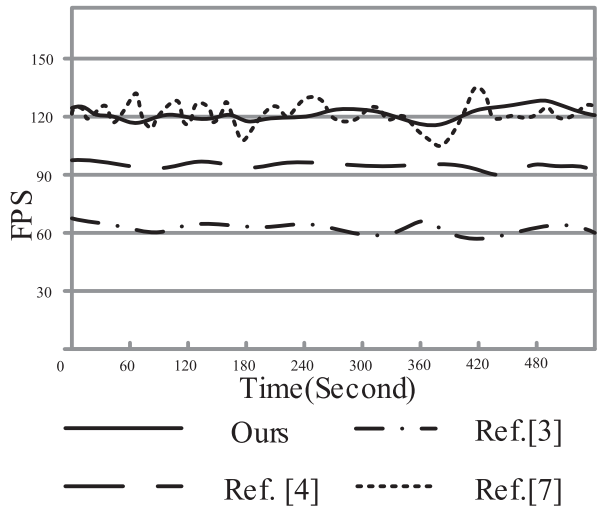

Fig. 8 Statistics of frame rate.

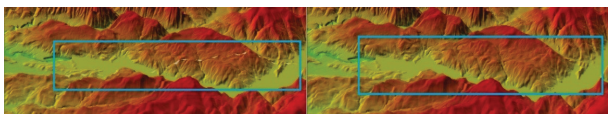

Fig. 9 Effect before and after eliminating cracks.

Fig. 10 Rendering results.

taking full advantage of GPU to achieve high rendering performance.

Finally, we do the experiment of cracks elimination with map data of Puget Sound whose colorful image is convenient for observation. Figure 9 gives the comparison of effect before and after eliminating cracks. We learn that cracks are eliminated perfectly by our method. Figure 10 shows some screenshots of real-time rendering by our method.

5. Conclusion and Future Work

We presented a novel real-time terrain rendering algorithm based on geometry clipmaps using new feathers such as hardware-supported tessellation available in modern GPUs. Since triangles are generated and rendered entirely on GPU, there is no need to transfer massive vertices from CPU. The experimental results shows that our algorithm improve rendering performance and stability of frame rate.

Our method can only be applied in Cartesian Coordinates for now. It would require our algorithm adaptive to spherical coordinate system (WGS84) for widely application.

Acknowledgments

This research is supported by the National High Technology Research and Development Program of China (863) (2015AA016404).

References

[1] R. Pajarola and E. Gobbetti, "Survey on semi-regular multiresolution models for interactive terrain rendering," The Visual Computer, vol.23, no.8, pp.583–605, 2007.
[2] F. Losasso and H. Hoppe, "Geometry clipmaps: Terrain rendering using nested regular grids," ACM Trans. Graph, vol.23, no.3, pp.769–776, 2004.
[3] A. Asirvatham and H. Hoppe, "GPU Gems 2 — Programming Techniques for HighPerformance Graphics and General-Purpose Computation," pp.27–46, 2005.
[4] A.M. Dimitrijević and D.D. Rančić, "Ellipsoidal Clipmaps-A planet-sized terrain rendering algorithm," Computers and Graphics, vol.52, pp.43–61, 2015.
[5] D. Feldmann, F. Steinicke, and K.H. Hinrichs, "Flexible Clipmaps for managing growing textures," International Conference on Computer Graphics theory and Applications, pp.173–180, 2011.
[6] D. Feldmann and F. Hinrichs, "GPU Based single-pass ray casting of large heightfields using clipmaps," Proceedings of compuer graphics international (CGI), 2012.
[7] E. Yusov and M. Shevtsov, "High-PerFormance Terrain Rendering Using Hardware Tessellation," Journal of WSCG, vol.19, no.3, pp.85–92, 2011.
[8] B.-Q. Zhang, et al., "Screen-space adaptive tessellation for terrain rendering," J. Image and Graphics, vol.17, no.11, pp.1431–1438, 2012.
[9] H.Y. Kang, H. Jang, C.-S. Cho, and J.H. Han, "Multi-resolution terrain rendering with GPU tessellation," The Visual Computer, vol.31, no.4, pp.455–469, 2015.