지형 렌더링을 위한 효율적인 자료 구조와 알고리즘

정 문 주† · 한 정 현††

요 약

대화적인 멀티미디어 시스템 구현에 있어, 실시간 가시화/시각화(visualization)는 중요한 기능을 한다. 본 논문은 실시간 지형 렌더링을 위한 효율적인 자료 구조와 알고리즘을 제안한다. 대개의 경우, 지형 데이터는 매우 방대한 크기를 가지고 있어서 있는 데이터를 그대로 실시간 렌더링하는 것은 불가능할 경우가 많다. 따라서 실시간 지형 렌더링에서는 LOD(Levels of Detail) 관리와 뷰 프러스텀 컬링이 핵심 사항이 된다. 본 논문은 계층적이면서도 간결한 지형 자료 구조, 신속한 뷰 프러스텀 컬링, 효율적인 LOD 구축 및 이에 기반한 렌더링 기법을 상세히 기술한다. 실험 결과, 제안된 기법은 일반 PC 사양에서 초당 22 프레임의 렌더링 속도를 보였다.

Efficient Data Structures and Algorithms for Terrain Data Visualization

Moon-Ju Jung† · JungHyun Han††

ABSTRACT

In implementing interactive multimedia systems, real-time visualization plays an important role. This paper presents efficient data structures and algorithms for real-time terrain navigation. Terrain data set is usually too huge to display as is. Therefore LOD (levels of detail) methods and view frustum culling are essential tools. This paper describes in detail compact hierarchical data structures, fast view frustum culling, and efficient LOD construction/rendering algorithms. Unlike previous works, we use a precise screen-space error metric for vertex removal and a strict error threshold allowing sub-pixel-sized errors only. Nevertheless, we can achieve 22 fps on average in a PC platform. The methods presented in this paper also satisfy almost all of the requirements for interactive real-time terrain visualization.

키워드 : 지형 렌더링(terrain rendering), LOD(Levels of Detail), 뷰 프러스텀 컬링(view frustum culling)

1. Introduction

Terrain rendering occupies an important part in several applications such as computer graphics, geographic information systems, games, virtual reality, flight simulation, etc. Such fields require terrain navigation or fly-through in an interactive real-time mode. However, the original terrain data are often too large to display at interactive frame rates despite the improvement of graphics hardware capabilities. Therefore, methods are needed which reduce the complexity of the terrain data but retain the image quality. The LOD (levels of detail) method has been adopted as a suitable tool that presents the near/important parts by a large number of small polygons, and the far/unimportant parts by a small number of large polygons.

Among several terrain data representations, DEM (digital elevation model) and TIN (triangulated irregular network) have been most popular. DEM, also called height filed, is a set of height/elevation data sampled in a regular grid. In contrast, the TIN is basically a triangulated or polygonalized mesh typically generated by extracting feature points from the height field data.

DEM-based LOD algorithms [2, 3, 5, 9, 14, 15, 18, 19, 25] usually construct a hierarchical structure such as quadtree [1, 20, 21], and dynamically remove and insert vertices. In contrast, TIN-based LOD algorithms [4, 6, 7, 13, 23, 24] usually adopt the traditional mesh simplification and LOD management techniques [8, 10-12, 22]. In general, TIN-based algorithms produce more optimal triangulation, but are not storage-efficient. In contrast, DEM-based algorithms have proven to be more effective in view frustum culling, and can easily manage LOD data structures. DEM-based algorithms have demonstrated higher performance, and have been preferred for terrain rendering [16].

The key issues in interactive real-time rendering of terrain data can be listed as follows.

- **View-dependent LOD :** DEM vertices far from the viewpoint may often be removed tremendously while near vertices often require a higher resolution.
- **Efficient view frustum culling :** This usually leads to

† 준 회 원 : 성균관대학교 대학원 정보통신공학부
†† 종신회원 : 성균관대학교 정보통신공학부 교수
논문접수 : 2002년 8월 13일, 심사완료 : 2002년 9월 27일

high performance improvement because, in a typical situation, only a small part of the large terrain is visible.

- **Accurate and efficient error metric** : An error metric determines whether a vertex can be removed without significant degradation of image quality.
- **Memory requirement minimization** : The terrain data are huge and therefore LOD algorithms with minimum memory requirement are preferred.
- **Localized update** : A change in the LOD representation should affect as small portions of the data as possible.
- **Popping minimization** : When the levels of detail are changed, there are often popping effects perceived, which should be minimized.

Among the issues listed above, the most important are LOD implementation and view frustum culling. This paper presents efficient data structures and algorithms to tackle the two issues. In addition, our approach provides satisfactory solutions to all the other issues listed above.

2. Related Work

This section reviews some recent DEM-based approaches related with our methods to be presented.

Lindstrom *et al.* [14] proposed a bottom-up traversal of the quadtree structure, where adjacent triangles are merged into a larger triangle by removing the shared vertex. For determining vertex removal, a *screen-space error metric* has been devised. They also used so-called *block-based simplification* for interactive frame rates. However, the devised error metric is not precise, and the LOD representation often contains many unnecessary triangles due to forced split of triangles for crack elimination.

Duchaineau *et al.* [5] proposed both top-down and bottom-up traverses with a binary triangle tree. *Dual queues* for split and merge of triangles are maintained, and *frame-to-frame coherence* can be exploited. However, like [14], they also suffer from forced split of triangles.

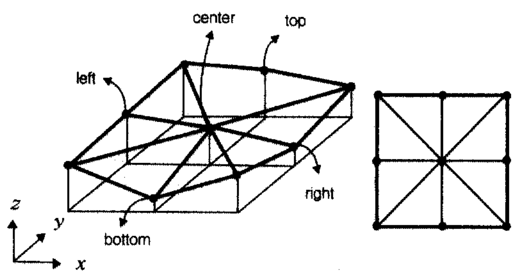
Röttger et al. [19] proposed a top-down traverse of the quadtree structure. The quadtree is *implicitly represented* by a matrix, where each element indicates whether the corresponding node of the quadtree exists or not. However, the resolution difference between adjacent triangles is restricted to one. This restriction often leads to unnecessarily many triangles, for example, when rendering a large flat terrain with a few local peaks.

Youbing *et al.* [25] proposed a simplified/fast view frustum culling where a subset of the frustum faces is projected onto the $z = 0$ plane and the visible area is roughly selected using the projection. However, their view frustum culling is too naive, and often leads to overly large part of the terrain data. Unlike forced-split methods, cracks are eliminated by suppressing the crack-causing vertices.

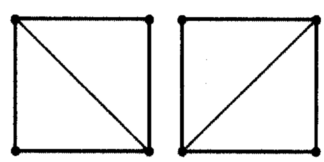
3. Hierarchical Structures of Height Field Data

3.1 Block and quadtree representation

This subsection briefly explains the height field representation proposed by Lindstrom *et al.* [14]. As depicted in (Figure 1) (a), the smallest representable mesh consists of 3×3 vertices, and is called a block. Out of the 9 vertices of a block, the *simplification* procedure considers 5 vertices (named **top**, **bottom**, **left**, **right** and **center**) as candidates for removal. If all of them are removed, we have either of the two possible triangulations shown in (Figure 1) (b),



(a) a block

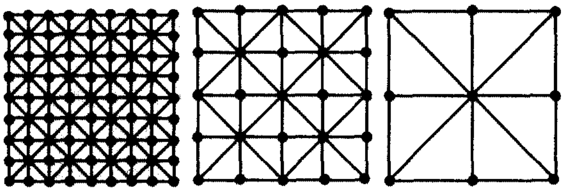


(b) completely simplified blocks

(Figure 1) a quadtree block and its simplification

Suppose that we are given an original mesh shown in (Figure 2) (a). It consists of 16 blocks. If all the 5 candidate vertices of each block are removed, the original mesh will be transformed into that of (Figure 2) (b). One more stage of such complete simplification will lead to the coarsest mesh in (Figure 2) (c).

This simplification strategy is compatible with the *quadtree* structure. Given $(2^k+1)\times(2^k+1)$ vertices, the quadtree will have k levels. Level 0 is the highest level and represents the coarsest mesh with 9 vertices ; level $(k-1)$ is the lowest level and represents the original mesh. (See the 3-level quadtree in (Figure 2)). A node of the quadtree corresponds to a block. Note that, except in the lowest level, a block in level $(m-1)$ has 4 child blocks in level m.

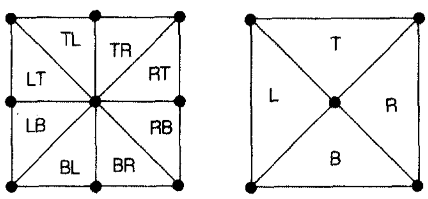


(a) level 2 (b) level 1 (c) level 0

(Figure 2) Quadtree levels

3.2 Block data structure

There can exist 12 triangles in a block, as depicted in (Figure 3) where **T** stands for Top, **B** for Bottom, **L** for Left, and **R** for Right. Before simplification, a block has the 8 triangles {**TL, TR, BL, BR, LT, LB, RT, RB**}. Some vertices (out of 5 candidates) may be removed for simplification. Suppose that, for example, the top vertex is removed. Then, two triangles **TL** and **TR** will be merged into a single larger triangle **T**, and the new triangulation of the block will be {**T, BL, BR, LT, LB, RT, RB**}.



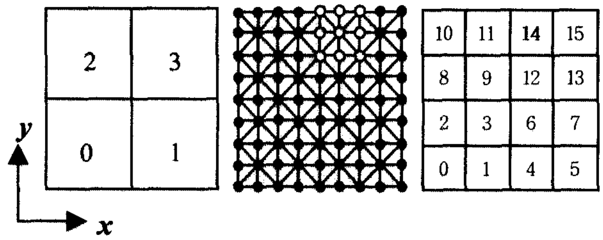
(Figure 3) 12 possible triangles in a block

To describe the status of a block, we use the data structure **Block** shown below. Each triangle in (Figure 3) is associated with a 1-bit flag, which indicates its presence/absence. Other fields of the **Block** structure will be discussed later.

```
struct Block
{
    unsigned TL : 1, TR : 1, BL : 1, BR : 1 ;
    unsigned LT : 1, LB : 1, RT : 1, RB : 1 ;
    unsigned T : 1, B : 1, L : 1, R : 1 ;
    unsigned visible : 1 ;
    WORD center_x, center_y ;
} ;
```

3.3 Block indexing

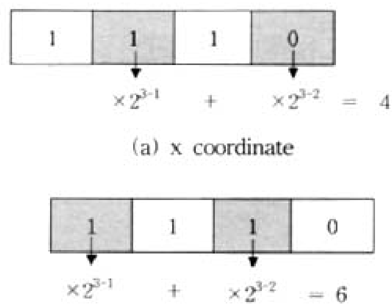
A block can be divided into 4 quadrants, and each quadrant is labeled as depicted in (Figure 4) (a) [20, 21]. With this labeling scheme, each block of the quadtree can be assigned an *index*. Consider the hollow-vertex block in (Figure 4) (b). It is positioned at the 3rd quadrant of the root node block, and so assigned code $11_{(2)}$. With respect to the 3rd quadrant, the hollow-vertex block is in the 2_{nd} quadrant, and so assigned code $10_{(2)}$. The block index is the concatenation of these quadrant codes : $1110_{(2)} = 14_{(10)}$. (Figure 4) (c) shows the indices of the 16 blocks of level 2.



(a) quadrant labels (b) a block in level 2 (c) block indices in level 2

(Figure 4) Block indices

The index of a block at level m consists of $2m$ bits. The parent-child relations between blocks can be easily found by shift operations : If an index of $2m$ bits is shifted right by 2 bits, the resulting $2(m-1)$ bits are the index of its parent block. For example, the hollow-vertex block $1110_{(2)}$ in (Figure 4) has the parent block $11_{(2)}$ at level 1. Thanks to this indexing scheme, the quadtree does not need pointers, and the LOD construction procedure (discussed in Section 5) can be implemented very efficiently.



(a) x coordinate

(b) y coordinate

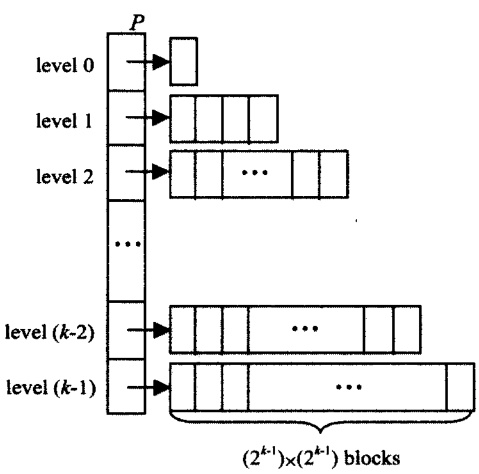
(Figure 5) Computing vertex coordinates with a block index

Given a block index, it is straightforward to compute the (x, y)-coordinates of the block vertices. Note that, for a $2m$-bit index, m consecutive 2-bit codes specify m quadtree blocks from levels 1 through m. Also note that a quadtree block at level m has the width/height of 2^{k-m} where k is the depth of the quadtree. Finally note that, in a 2-bit code, the first bit determines the offset along y-axis and the second bit determines x-offset, where the offset is either 0 or the width/height of the associated block. These 3 facts allow us to compute the x-coordinate of a block's lower-left vertex by multiplying each second bit (of a block index) by the width/height of the associated block and then adding them up. Similarly, we can obtain the y-coordinate by processing every first bit. (Figure 5) demonstrates how to compute the (x,y)-coordinates of the lower-left vertex of the example block in (Figure 4) (b). Both the x- and y-coordinates of the lower-left vertex are incremented by half of the width/height of the associated block, and then assigned to **center_x** and **center_y** of the **Block** structure.

3.4 Overall Data Structures

A separate array of **Block** structures is maintained for a quadtree level, and therefore we have k arrays. The pointers to these k arrays are stored in another array P as shown in (Figure 6). Given a level number l and a block index i, its **Block** structure can be directly accessed using $*(*(P+l)+i)$.

We also have two 2D arrays : one for storing height/z values and the other for **Vertex** structures (which will be discussed in Section 5.1). Indices of these two arrays are equivalent to the (x, y)-coordinates of DEM vertices, and therefore the data are directly accessible using the vertex (x,y)-coordinates.



(Figure 6) Arrays of Block structures

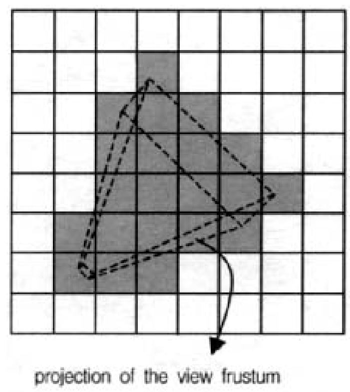
4. View Frustum Culling

Efficient view frustum culling plays a key role in real-time rendering. However, traditional 3D view frustum culling incurs a significant amount of computational cost. Youbing *et al.* [25] proposed to project the view frustum onto the xy-plane, generate a bounding triangle for the projected view frustum, and then render the blocks which intersect the triangle. We take a similar but more elaborate approach.

Eight vertices of the view frustum are projected onto the z = 0 or xy plane, and their *convex hull* is computed. It is then checked whether the convex hull and the root node block of the quadtree intersect. The block's xy-range is computed using the level number and **(center_x, center_y)** of the **Block** structure.

Recall that the root node of the quadtree corresponds to the entire height field, and its 4 children (at level 1) are the 4 quadrants of the entire height filed. If the convex hull and the root node intersect, the root node's 4 child nodes/blocks are visited and tested for intersection with the convex hull. This *top-down* procedure of intersection test is done recursively. If a block does not intersect the convex hull, the recursion stops and its children are not visited. For each lowest level block which intersects the convex hull, the **visible** flag is set to 1.

(Figure 7) shows the 'visible' blocks obtained through the topdown recursive view frustum culling. Note that the 'visible' blocks are not guaranteed to be inside the view frustum, but simply candidates with which the LOD representation will be constructed.



(Figure 7) The lowest-level blocks intersecting the convex hull

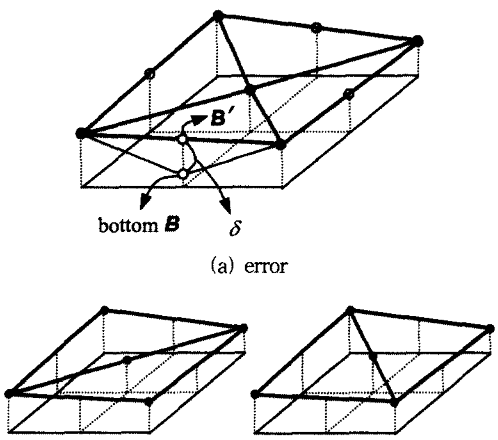
5. LOD Construction and Rendering

5.1 Error metric for vertex removal

As discussed in Section 3.1, a block has 5 candidate vertices for removal. (Figure 8) (a) shows a simplified block where **top, bottom, left** and **right** vertices are removed from (Figure 1) (a). Note that, when any of the 4 vertices is removed, its height becomes the average of its neighboring vertices' heights, as illustrated in (Figure 8) (a) where δ denotes the error caused by removing the **bottom** vertex.

Projection of δ onto the screen determines the *screen-space error* δ_s. If δ_s is smaller than a pre-defined threshold τ, the vertex can be safely removed. This is exactly what Lindstrom *et al.* [14] proposed. For the sake of simplicity in computing δ_s, however, Lindstrom *et al.* made strong assumptions which are not reasonable(See [14] for details). Instead, we use the OpenGL utility function **gluProject ()**

which maps object coordinates to screen spaces : **gluProject ()** is invoked with $\boldsymbol{B}$ and $\boldsymbol{B'}$ respectively and returns their screen-space coordinates[17]. Then, the distance between the two screen-space points is δ_s. If $\delta_s < \tau$, $\boldsymbol{B}$ is removed. Unlike Lindstrom *et al.*, we compute the precise error. However, the computing time turns out a little faster than the method by Lindstrom *et al.* because **gluProject ()** ultimately relies on hardware acceleration.



(a) error

(b) 2 possible triangulations

(Figure 8) Vertex simplification and resulting blocks

As δ is a constant per a vertex, it is computed at the preprocessing stage. For each frame, δ is retrieved and ($\boldsymbol{B}$, $\boldsymbol{B'} = \boldsymbol{B} + \delta$) is used to compute δ_s through **gluProject ()**. Note that storing δ (instead of $\boldsymbol{B'}$) is more storage-efficient.

When the center vertex is removed from (Figure 8) (a), two triangulations can be made with the remaining 4 vertices. If the block corresponds to either the 0th or the 3rd quadrant of its parent, we will have the first configuration in (Figure 8) (b). If either 1st or 2nd, we have the second configuration.

Each vertex of the original DEM is represented by the following **Vertex** structure. The 1-bit flag **check** is set to 1 once the screen-space error δ_s has been computed for the vertex. Without **check** flag, δ_s would be computed twice because each of the top, bottom, left and right vertices is shared by two blocks. If δ_s leads to the vertex removal, the **important** flag is set to 0. Otherwise, it is set to 1. The **delta** flag stores δ computed at the preprocessing stage.

```
struct Vertex
{
        unsigned check : 1 ;
        unsigned important : 1 ;
        float delta ;
} ;
```

5.2 LOD construction in a bottom-up mode

The initialization stage sets all of 13 flags (for triangles and visibility) of the **Block** structure to 0, and the **check** and **important** flags of the **Vertex** structure to 0 and 1, respectively.

Then, view frustum culling is invoked to traverse the quadtree in a top-down mode. As a result, a subset of the lowest level blocks will have **visible** flags set to 1. For those blocks, **TL, TR, BL, BR, LT, LB, RT,** and **RB** are set to 1, and **T, B, L** and **R** are set to 0, i.e. only the original 8 triangles are made *active*.

LOD construction is done in a bottom-up mode. It visits the lowest level blocks with **visible** flag set to 1. Each block has 5 candidate vertices for removal : **top, bottom, left, right** and **center**. Because the **center** vertex is removable only after all of the other 4 vertices are removed, simplification is done in two stages : {**top, bottom, left, right**} first, and then {**center**}.

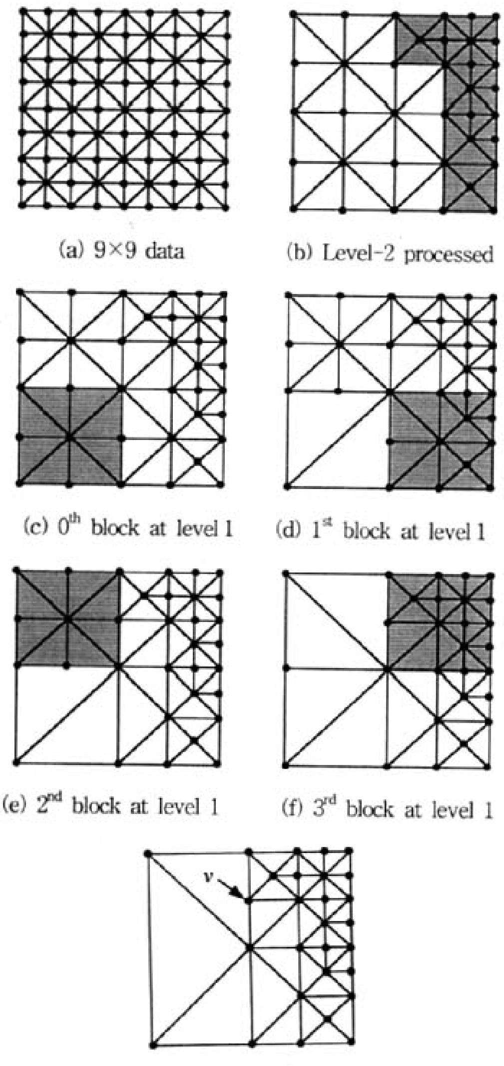
Any vertex in {**top, bottom, left, right**} can be removed only when two triangles sharing the vertex exist in the block. For example, removal of the **top** vertex requires the existence of **TL** and **TR** triangles (See (Figure 1) and (Figure 3)). It might seem that such a condition always holds. However, it does not always do as will be demonstrated later.

When the above condition holds, the **check** flag of the **Vertex** structure is checked to see if the vertex was already tested for removal when the adjacent block was processed. If the check **flag** is 1, we need to check only the **important** flag. If it is 0, the vertex will be removed. Otherwise, the vertex will remain alive and contribute to the final image.

If the **check** flag of the **Vertex** structure is 0, δ_s will be computed using **gluProject ()**. If $\delta_s < \tau$, the vertex will be removed. For the adjacent block processing, the **check** flag is then set to 1 and the **important** flag is set to 0. If δ_s, the vertex is not removed, and both the **check** and **important** flags are set to 1.

When a vertex is removed, related triangle flags should be changed appropriately. If the **top** vertex is removed, for example, the **TL** and **TR** flags are set to 0 and T flag is set to 1.

After the vertex set {**top, bottom, left, right**} is processed, the **center** vertex is tested for removal. Note that, if the center vertex is removed, the two resulting triangles (in either of the triangulations in (Figure 8) (b)). do not belong to the current block any longer. Instead, they belong to the parent block. Therefore the **visible** flag of the current block is set to 0, and the **visible** flag of the parent block is set to 1. This means that rendering is done with the corresponding quadrant of the parent block, not with the current block. Simultaneously, the corresponding triangle flags of the parent block are set to 1. If the current block corresponds to the 0^{th} quadrant of the parent block, for example, **LB** and **BL** will be set to 1. (They were set to 0 by the initialization stage.)



(a) 9×9 data (b) Level-2 processed

(c) 0^{th} block at level 1 (d) 1^{st} block at level 1

(e) 2^{nd} block at level 1 (f) 3^{rd} block at level 1

(g) Final LOD representation

(Figure 9) LOD construction for $(2^3+1)\times(2^3+1)$ - sized height field

Note that, if a block's 5 vertices are all removed, its parent block's **visible** flag is set to 1. The recursive bottom-up procedure of LOD construction visits all blocks with the **visible** flag set to 1. (Figure 9) shows the step-by-step procedure of LOD construction with (2^3+1) (2^3+1) - sized data. The data lead to 3-level quadtree. Suppose that all of the 16 blocks at level 2 (the lowest level) are visible. Also suppose that (Figure 9) (b) is the result of applying vertex removal operations to the 16 level-2 blocks. We can see that the 5 shaded blocks' **visible** flags are 1 while the remaining 11 blocks' **visible** flags are 0.

As level-2 block processing is completed, level 1 will be processed. According to the above **visible** flag setting procedure, the 4 level-1 blocks' **visible** flags have been set to 1. (Figure 9) (c) through (Figure 9) (f) show the sequence of block visits. Suppose that all of 5 vertices are removed from the 0^{th} block at level 1. (Figure 9) (d) shows the result. Then, the 0^{th} block's **visible** flag is set to 0, and the parent (level-0) block's **visible** flag is set to 1.

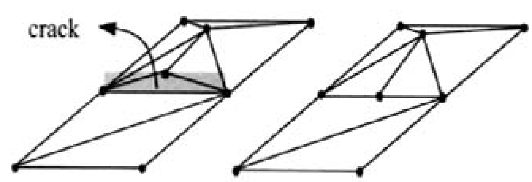
Visited next is the 1^{st} block at level 1. Note that, in (Figure 9) (d), only 4 triangles (TL, LT, LB, and BL) exist in this block. Also recall that a vertex in {**top, bottom, left, right**} can be removed only when two triangles sharing the vertex exist in the block. Therefore, the **left** vertex is the only candidate for removal in the 1^{st} block. Suppose it is removed as its δ_s is smaller than τ. The result is shown in (Figure 9) (e).

Suppose all 5 vertices of the 2^{nd} block are removed. (Figure 9) (f) shows the result. In the 3^{rd} block, only two triangles (LB and BL) exist and they do not share any vertex in {**top, bottom, left, right**}. Therefore, no vertex can be removed in this block.

(Figure 9) (f) is the result when all level-1 blocks are processed. The **visible** flag of the level-0 block has been set to 1, and so the block is visited for further simplification. As shown in (Figure 9) (f), only the **left** vertex is a candidate for removal as both LT and LB exist. If the vertex is removed, the result will be that of (Figure 9) (g).

5.3 Crack elimination

Before rendering the LOD representation, *crack elimination* should be done. In (Figure 9) (g), vertex v causes a crack. Our approach to crack elimination is the same as that by Youbing *et al.* [25]. We simply suppress the vertex that causes a crack as illustrated in (Figure 10). This is reasonable in that such a crack-causing vertex has a screen-space error δ_s smaller than τ. Despite this fact, the vertex remains in the final LOD representation only for valid triangulation of the mesh. Notice that the crack-causing vertex simply needs to exist for valid triangulation but does not have to maintain the original height.



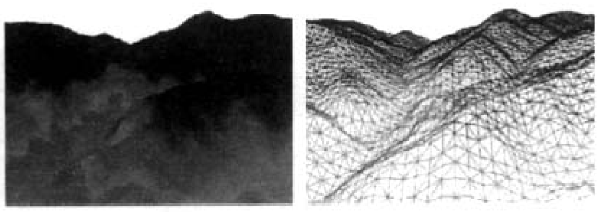
(Figure 10) Crack elimination by height adjustment

5.4 Rendering of the LOD data

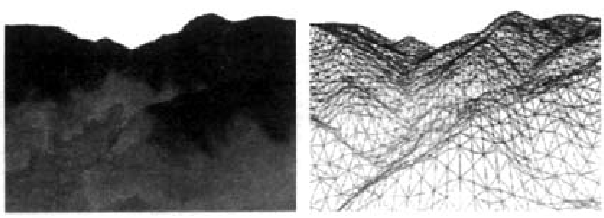
Rendering can be done either by top-down traverse or by bottom-up traverse of the quadtree. Starting from either the highest level or the lowest level, we visit only the blocks with **visible** flags set to 1. In each visited block, we render only the triangles whose flags are 1. After processing all blocks in the level, recursively visit the next levels until all active triangles are rendered.

For rendering the next frame, all active blocks and triangles are deactivated immediately after being rendered, i.e. their flags are reset to 0. The next frame rendering starts by view frustum culling.

6. Implementation Results

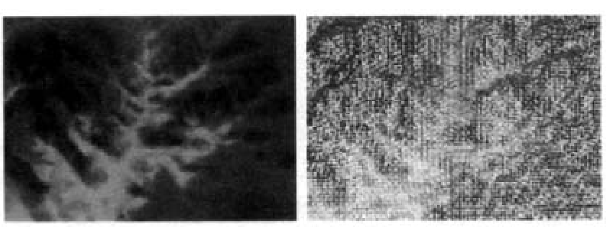


(a) Rendering using original data

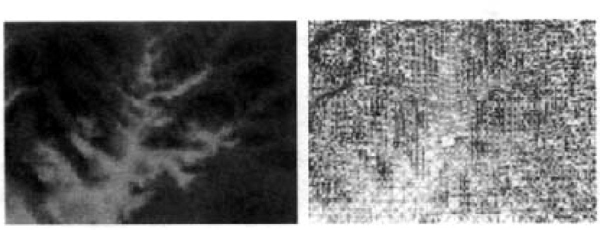


(b) Rendering using LOD representation

(Figure 11) 513×513 data rendering example 1



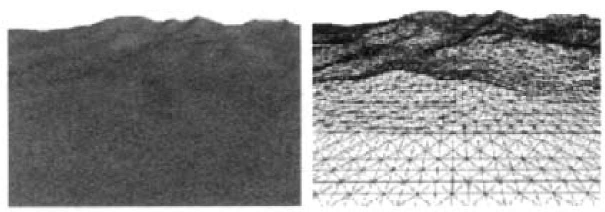
(a) Rendering using original data



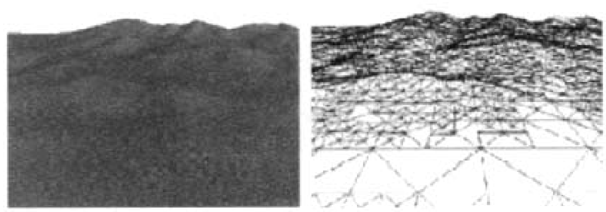
We have implemented and tested our approach on Pentium3 866 PC with 384M RAM and NVIDIA GeForce2 MX card. Coding is done using Visual C++ and OpenGL in Windows environment. (Figure 11) and (Figure 12) compare texture-mapped rendering results using the original 513×513-sized DEM data and the LOD representation.

The threshold τ is set to 0.5, which means that only sub-pixel-sized errors are allowed. You can notice that there are little differences between two images in both figures. (Figure 13) compares rendering results of 1025×1025 - sized data and the LOD representation. The threshold τ is also set to 0.5 (No texture map is available for this 1025×1025 data set.).

Rendering performance is analyzed for 100 frames using the 513×513 data, and depicted in (Figure 14). The average frame rate is 22(fps), and the average number of triangles is 7283.



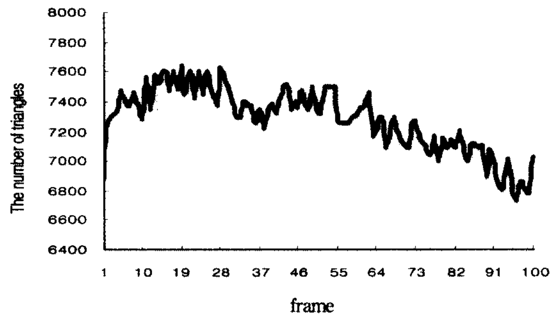
(a) Rendering using original data



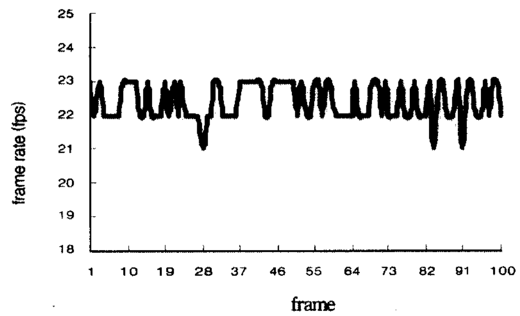
(b) Rendering using LOD representation

7. Conclusion

We have presented efficient LOD data structures and algorithms for interactive real-time rendering of terrain data. Using an inherited indexing scheme, we can achieve very compact data structures. Efficient algorithms for view frustum culling and bottom-up LOD construction are also presented. Unlike the previous works, we use a precise screen-space error metric for vertex removal. With the threshold set to 0.5, we can virtually achieve error-free and popping-free rendering. Despite this strict error threshold, 22 fps is achieved on average. The methods presented in this paper also satisfy almost all of the requirements for interactive real-time terrain visualization.



(a) Numbers of triangles per frame



(b) Frame rates

(Figure 14) Performance Analysis

References

[1] Balmelli, L., Kovačević, J., and Vetterli, M., "Quadtrees for Embedded Surface Visualization : Constraints and Efficient Data Structures," *Proceedings of IEEE International Conference on Image Processing (ICIP)*, Vol.2, pp.487-491, Oct., 1999

[2] Blow, J., "Terrain Rendering at High Levels of Detail," *Proceedings of the 2000 Game Developers Conference*, Mar., 2000.

[3] Castle, L., Lanier, J., and McNeill, J., "Real-time Continuous Level of Detail (LOD) for PCs and Consoles," Technical Presentation GDC, 2000.

[4] De Berg, M., and Dobrindt, K. T. G., "On Levels of Detail in Terrains," *11th ACM Symposium on Computational Geometry*, Jun., 1995.

[5] Duchaineau, M. A., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., and Mineev-Weinstein, M. B., "ROAMing Terrain : Real-time Optimally Adapting Meshes," *IEEE Visualization '97*, pp.81-88, Nov., 1997.

[6] Ferguson, R. L., Economy, R., Kelly, W. A., and Ramos, P. P., "Continuous Terrain Level of Detail for Visual Simulation," *Proceedings IMAGE V Conference*, pp.144-151, Jun., 1990.

[7] Garland, M., and Heckbert, P. S., "Fast Polygonal Approximation of Terrains and Height Fields," Technical Report CMU-CS-95-181, CS Dept., Carnegie Mellon U., 1995.

[8] Garland, M., and Heckbert, P., "Surface Simplification Using Quadric Error Metrics," *Proceedings of SIGGRAPH '97*, pp.209-216, Aug., 1997.

[9] Gross, M., Gatti, R., and Staadt, O., "Fast Multiresolution Surface Meshing," *IEEE Visualization '95*, pp.135-142, Oct., 1995.

[10] Heckbert, P. S., Garland, M., "Multiresolution Modeling for Fast Rendering," *Proceedings of Graphics Interface '94*, pp.43-50, May, 1994.

[11] Hoppe, H., "Progressive Meshes," *Proceedings of SIGGRAPH '96*, pp.99-108, Aug., 1996.

[12] Hoppe, H., "View-Dependent Refinement of Progressive Meshes," *Proceedings of SIGGRAPH '97*, pp.189-198, Aug., 1997.

[13] Hoppe, H., "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering," *IEEE Visualization '98*, pp.35-42, Oct., 1998.

[14] Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., and Turner, G. A., "Real-Time, Continuous Level of Detail Rendering of Height Fields," *Proceedings of SIGGRAPH '96*, pp.109-118, Aug., 1996.

[15] Lindstrom, P., and Pascucci, V., Visualization of Large Terrains Made Easy, *IEEE Visualization 2001*, pp.363 -370, Oct., 2001.

[16] Ogren, A., Continuous Level of Detail in Real-Time Rendering, Master's Thesis, 2000.

[17] OpenGL Architecture Review Board, *OpenGL Reference Manual*, Addison-Wesley, 2000.

[18] Pajarola, R. B., Large Scale Terrain Visualization Using the Restricted Quadtree Triangulation, *IEEE Visualization '98*, pp.19-26, Oct., 1998.

[19] Röttger, S., Heidrich, W., Slussallek, P., and Seidel, H. -P., Real-Time Generation of Continuous Levels of Detail for Height Fields, *Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization*, pp.315-322, Feb., 1998.

[20] Samet, H., The Quadtree and Related Hierarchical Data Structures, *ACM Computing Surveys*, Vol.16, No.2, pp.187-260, Jun., 1984.

[21] Samet, H., *Applications of Spatial Data Structures : Computer Graphics, Image Processing, and GIS*, Addison-Wesley, 1989.

[22] Schroeder, W. J., Zarge, J. A., and Lorenson, W. E., Decimation of Triangle Meshes, *Proceedings of SIGGRAPH '92*, pp.65-70, Jul., 1992.

[23] Xia, J. C., and Varshney, A., Dynamic View-Dependent Simplification for Polygonal Models, *IEEE Visualization '96*, pp.327-334, 1996.

[24] Xia, J. C., El-Sana, J., and Varshney, A., Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models, *IEEE Transactions on Visualization and Computer Graphics*, Vol.3, No.2, 1997.

[25] Youbing, Z., Ji. Z., Jiaoying, S., and Zhigeng, P., A Fast Algorithm for Large Scale Terrain Walkthrough, *CAD& Graphics 2001*, Aug., 2001.

정 문 주

e-mail : lunar36@freechal.com

2001년 성균관대학교 정보통신공학부 (학사)

2001년~현재 성균관대학교 정보통신공학부 석사

관심분야 : 컴퓨터 그래픽스

한 정 현

e-mail : jhan@skku.ac.kr

1988년 서울대학교 컴퓨터공학과(학사)

1991년 Univ. of Cincinnati Computer Science(공학석사)

1996년 Univ. of Southern California Computer Science(공학박사)

1996년~1997년 National Institute of Standards and Technology(NIST)

1997년~현재 성균관대학교 정보통신공학부 조교수

관심분야 : 컴퓨터 그래픽스, CAD/CAM 등