IMPROVEMENT ON DYNAMIC TILED TERRAIN RENDERING ALGORITHM IN LIBMINI

Liao Mingxue[(a)], Xu Fanjang[(a)], He Xiaoxin[(a)]

[(a)]Institute of Software, Chinese Academy of Sciences

[(a)]liaomingxue@sohu.com

ABSTRACT

As an open source library for large-scale terrain rendering in a continuous LOD high field, libMini takes a top-down method for static terrain rendering and achieves good performance. However, it uses the same top-down method to render elevation and texture for dynamic terrain. The method consumes a few seconds to update dynamic terrain and thus does not meet real-time requirements for rendering and leads to possible holes and gaps between adjacent terrain tiles. A millisecond-level real-time bottom-up libMini-based algorithm is proposed to render dynamic terrain while a method is presented to blend holes and gaps produced during the process of rendering dynamic tiled terrain.

Keywords: bottom-up algorithm, dynamic terrain, libMini, tiled terrain

1. INTRODUCTION

TERRAIN visualization is an important part of geographic information system, virtual battlefield environment, simulating training system, games and so on. Many algorithms are published for static terrain visualization, such as real-time continuous level of detail rendering algorithm (Lindstrom 1996), progressive meshes algorithm (Hoppe 1996), ROAM (Duchaineau 1997) and view-dependent fast real-time generating algorithm for large-scale terrain (Jin 2009). Due to the development of virtual environment simulation and increase in requirements for real-time interaction, the research on dynamic terrain visualization algorithms becomes increasingly important.

Both elevation data and image data of terrain will be changed due to interactions between 3D models and terrain in battlefield simulation or other situations, for example, explosion of thousands of bombs will force ground surfaces to collapse and will expose deep soil that takes on a different appearance from the ground surfaces and thus will pose restriction on the movement of tanks and other models in the scene. A few methods are now invented for dynamic terrain rendering.

Robert (1999) described a model of ground surfaces and explained how these surfaces can be deformed by characters in an animation. But their simulation model of ground surfaces was based on a uniform-resolution height field that cannot be extended to large-scale terrain scene.

Shamir proposed a multi-resolution dynamic meshes algorithm, which concentrates on complex geometric objects other than large-scale terrain (Shamir 2000). They used DAG (Directed Acyclic Graph) to present hierarchical structure and updated the DAG as deformations happened on objects at time steps to form T-DAG. This relatively costly T-DAG updating has a limit to on-line modifications on objects.

With an extension to ROAM and using DEXTER (Dynamic EXTEnsion of Resolution), He provided an algorithm for dynamic terrain visualization (He 2002). But their approach united with fake properties of terrain deformation and did not consider the physical model of terrain. They only dealt with relatively small scale terrain. Their terrain was divided into regions, but the continuity among the regions was not processed.

Recently, based on ROAM, Cai implemented a dynamic terrain method for rendering craters in battle-field environments (Cai 2006), but they did not address how the method runs smoothly in real-time. Exploiting the power of modern GPU, Shiben developed a system for real-time rendering and manipulation of large terrains (Shiben 2008). Their system achieves a performance of 250-microsecond terrain deformation over a block of size 1024×1024. However, the performance can be only reached with the help of GPU and thus costly.

This paper firstly proposes a real-time dynamic terrain algorithm for large-scale terrain based on libMini library (Stefan 1998). The library achieves good LOD continuity and rendering effect and has been applied well in VTP (Discoe 2005, Discoe 2009) and AquaNox game (Stefan 1998). In the library, the terrain is divided into tiles to solve the problem in large-scale data rendering. This paper also implements a dynamic terrain algorithm for elevation and image updating in the tiled terrain to eliminate holes and gaps between adjacent tiles.

2. BRIEF INTRODUCTION TO ALGORITHM OF LIBMINI

The underlying data structure of libMini algorithm for terrain rendering is basically a quadtree. The quadtree is represented by a Boolean matrix. Beginning from the

root node of the tree, if the terrain block represented by a parent node needs to upgrade its rendering detail level, we set corresponding value in the matrix to 1 and continue this way with the parents' four subnodes. A decision variable *f* with a less-than-one value in (1) tells a node to be upgraded (Stefan 1998).

$$f = \frac{l}{d \cdot C \cdot \max(c \cdot d2, 1)} \quad (1)$$

$$d2 = \frac{1}{d} \max_{i=1\ldots6} |dh_i| \quad (2)$$

In both (1) and (2), *l* is the distance to eye point, *d* is the length of the terrain block presented by the node, and *d2* is the roughness of the block. The constant *C* determines the minimum global resolution, whereas the constant *c* specifies the desired global resolution. The dh_i, which also appear in Figure 1 (Stefan 1998), are the absolute values of the differences between center elevation and average elevation of two ends of four borders and two diagonals.

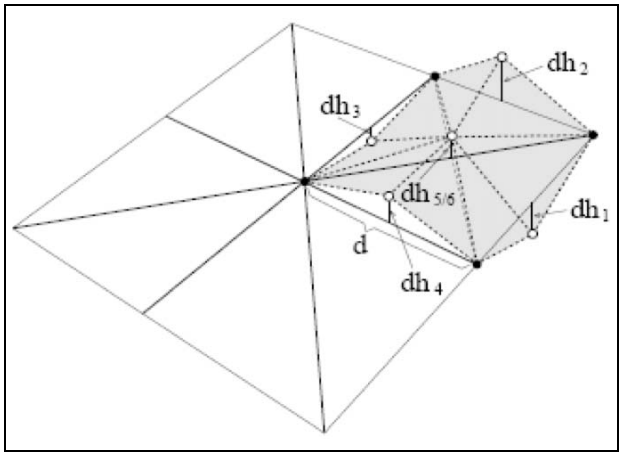

Figure 1: Definition for Variables dh_i in (2)

The major issue in rendering terrain is how to guarantee that the level difference of adjacent blocks is not greater than one in order to build a continuous mesh without holes. LibMini takes a top-down algorithm as in Figure 2 (Yang 2009) starting with highest-resolution blocks, it calculates their *d2*-values and then propagates those values to lower-resolution blocks to decide what real *d2* values of the lower-resolution blocks are.

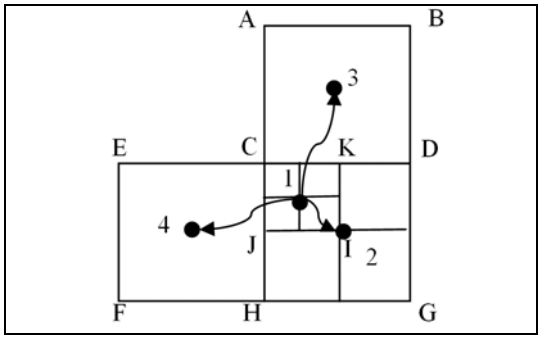

Figure 2: Process of Propagation of *d2*-values

As in Figure 2 the K (Stefan 1998) times *d2*-value is propagated from higher-resolution block CKIJ to lower ones ABCD, CEFH, and CDGH. Generally, a block in higher-resolution will propagate K times its *d2*-value to its parent block in lower-resolution and two lower-resolution blocks adjacent to its parent block. The final *d2* value of a block is the maximum of its own *d2* value and all K-times *d2* values passed to it.

According to the top-down algorithm above, if a small local part of the whole terrain changes dynamically, libMini need to calculate the new *d2*-value of the highest-resolution blocks in the local part and then to propagate the new values to lower-resolution blocks around this part. And these lower-resolution blocks also need to propagate their new *d2*-values to much-lower-resolution blocks around them. And such propagations will be continued this way up to the lowest-resolution blocks. In conclusion, a little change to any smallest part of terrain will trigger a continuous change to large-scale part of the terrain. Such changes are very time costly especially in a frequently changed terrain environment. So a smart algorithm must be designed to avoid such costly operation on dynamic terrain.

3. ALGORITHM FOR UPDATING BOTH ELEVATION AND TEXTURE

3.1. Elevation Updating

When a piece of terrain changes dynamically, the elevation of the terrain will usually be changed. Then, the changed terrain can act on the moving 3D models, such as tanks.

There are three steps to update elevation. Firstly, we use libMini API to get the highest-resolution elevation data of the changed terrain. Then, we modify the necessary part of the elevation data to demonstrate some dynamic terrain effect such as craters caused by sudden bomb explosion. At last, we build elevation data in lower resolutions with a simple interleaved method shown in Figure 30.

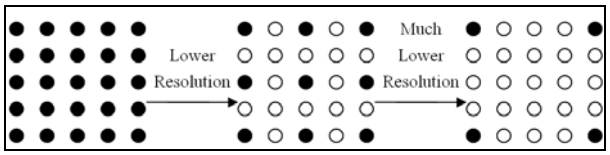

Figure 3: Simple Interleaved Method to Build Lower-resolution Data From Higher-resolution Data

After updating elevation we need to recalculate the *d2* values before rendering the new elevation. LibMini recalculates all *d2* values within about 2~3 seconds for 2048*2048 grids so that it cannot reach a real-time performance. Naturally we hope to only update a small local region where elevation is changed. However, libMini does not support such operation because using the top-down algorithm to propagate *d2* values in a local region will lead to incorrect *d2* values for lower-resolution blocks in other regions. For example, if the

old final *d2* value *x* of block CDGH in Figure 2 is set to K times the old *d2* value *y* of block CKIJ and the new *d2* value *z* of CKIJ is smaller than *y*, then the new final *d2* value of CDGH will not be updated because *x*>K*z*. In fact, if K*z* is the maximum value of all values passed to CDGH, the new final *d2* value of CDGH should have been updated to K*z*.

In conclusion, the top-down algorithm can be efficiently used for elevation update in the whole region, but not effective in a local region. Here, we devise a 2-step bottom-up algorithm to solve this problem.

Firstly all *d2* values of highest-resolution blocks in the changed local region are calculated according to a mathematical model.

Second, based on the rule of *d2*-value propagation, by a simple process of check for all higher-resolution blocks related with a lower-resolution block, it can be deduced that the *d2* value of a lower-resolution block is affected only by 12 blocks in higher resolution. As in figure 4, by checking higher-resolution blocks around block CDGH, we know its *d2*-value is affected by *d2*-values of blocks: A'K'KC, K'B'DK, DD'LI', LG'GI', GNMM', MH'HM', HF'J'J, JJ'CE', CKIJ, KDI'I, II'GM', JIM'H. The general situation is shown in figure 5 (Yang 2009). Then, a bottom-up algorithm for updating *d2*-values can be devised as below.

Bottom-up Algorithm
for each resolution ***r*** *from lower to higher*
 Evaluate block scope that need updating ***d2*** *value*
 if ***r*** *is the highest-resolution*
  for each block ***b*** *that need updating* ***d2***
   With (2) to calculate ***d2*** *of* ***b****, noted by* ***b.d2***
  endfor
 else
  for each block ***b*** *that need updating* ***d2***
   calculate ***b.d2***
   calculate $Kd2_1$, $Kd2_2$, …, $Kd2_{12}$
   $b.d2 \leftarrow max\{b.d2, Kd2_1, \ldots, Kd2_{12}\}$
  endfor
 endif
end for

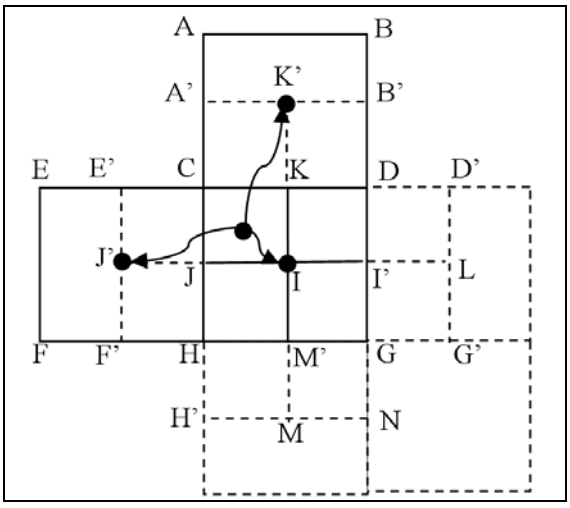

Figure 4: A Process of Finding Which *d2*-values Will Affect *d2*-value of CDGH Block

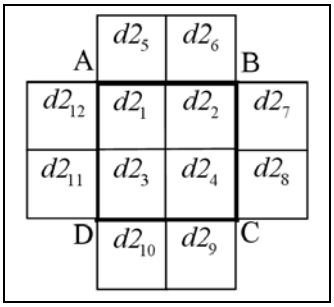

Figure 5: All 12 *d2*-values That Will Determine Lower-resolution Block ABCD's *d2* Value

For correctness of the bottom-up algorithm, we should prove a theorem given below.

Theorem 1. *The top-down algorithm in libMini for spreading* ***d2*** *values is equivalent to the bottom-up algorithm for spreading* ***d2*** *values.*

Proof. Firstly we build a discrete coordinate system with two coordinate axes ***I*** and ***J***. The starts of both ***I*** and ***J*** are 0. The ends of them are 2^n. The start point $<\boldsymbol{i}, \boldsymbol{j}>$ of blocks with resolution $2^k (\boldsymbol{n} \geqslant \boldsymbol{k} \geqslant 0)$ must be multiples of 2^k. The coordinates of a block with 2^k resolution are expressed by $[<\boldsymbol{i}, \boldsymbol{j}>, <\boldsymbol{i}+2^k, \boldsymbol{j}+2^k>]$. Figure 6 demonstrates such a coordinate system where0 the coordinate system starts with point <0, 0>, ends with point <4, 4>. The minimum resolution of blocks is 1. The maximum is 4. Figure 6 shows 16 blocks with minimum resolution, 4 blocks with 2^1 resolution, and 1 block with with 2^2 resolution. The 4 blocks with 2^1 resolution are [<0,0>,<2,2>], [<2,0>,<4,2>], [<0,2>,<2,4>] and [<2,2>,<4,4>], they all start with multiples of their resolution 2^1. Then, a 2^k resolution block may be in 4 different possible types of position as shown in (3) to (6). The 4 different types of position are shown by blocks A, B, C, D in figure 6.

$$I \equiv 0 \bmod 2^{k+1} \wedge J \equiv 0 \bmod 2^{k+1} \quad (3)$$
$$I \equiv 2^k \bmod 2^{k+1} \wedge J \equiv 0 \bmod 2^{k+1} \quad (4)$$
$$I \equiv 0 \bmod 2^{k+1} \wedge J \equiv 2^k \bmod 2^{k+1} \quad (5)$$
$$I \equiv 2^k \bmod 2^{k+1} \wedge J \equiv 2^k \bmod 2^{k+1} \quad (6)$$

Based on the top-down algorithm in libMini, the *d2* values are spread from a higher-resolution block to three lower-resolution blocks. We take a notation *top-down(b)* to indicate the blocks set containing blocks to which a higher-resolution block *b* $[<I, J>, <I+2^k, J+2^k>]$ spreads *d2* value. If the block *b* is in position (3), then the *top-down(b)* set consists of three elements as below:

(3)→*top-down(b)*=
{
$[<I, J>, <I+2^{k+1}, J+2^{k+1}>]$, (7)
$[<I, J-2^{k+1}>, <I+2^{k+1}, J>]$, (8)
$[<I-2^{k+1}, J>, <I, J+2^{k+1}>]$ (9)
}

The other three cases of *top-down(b)* are as below.

(4)→*top-down(b)*=
{
$[<I-2^k, J>, <I+2^k, J+2^{k+1}>]$, (10)
$[<I-2^k, J-2^{k+1}>, <I+2^k, J>]$, (11)
$[<I+2^k, J>, <I+2^k+2^{k+1}, J+2^{k+1}>]$ (12)
}
(5)→*top-down(b)*=
{
$[<I, J-2^k>, <I+2^{k+1}, J+2^k>]$, (13)
$[<I, J+2^k>, <I+2^{k+1}, J+2^k+2^{k+1}>]$, (14)
$[<I-2^{k+1}, J-2^k>, <I, J+2^k>]$ (15)
}
(6)→*top-down(b)*=
{
$[<I-2^k, J-2^k>, <I+2^k, J+2^k>]$, (16)
$[<I+2^k, J-2^k>, <I+2^k+2^{k+1}, J+2^k>]$, (17)
$[<I-2^k, J+2^k>, <I+2^k, J+2^k+2^{k+1}>]$ (18)
}

For any lower-resolution block $b[<I, J>, <I+2^M, J+2^M>]$, it must be a member of top-down set of a certain higher-resolution block. Let $b[<I', J'>, <I'+2^M, J'+2^M>]$ be equal to any of (7)~(18), we can get all possible higher-resolution block *B* that propagate their *d2* values to *b*. For example, let *b* be equal to (7), we have:

$$[<I',J'>,<I'+2^M,J'+2^M>]=[<I,J>,<I+2^{k+1},J+2^{k+1}>]. \quad (19)$$

From (19), we can know the relations below:

$$I=I',\ J=J',\ k=M-1. \quad (20)$$

Then *b* becomes:

$$b=[<I',J'>,<I'+2^{M-1},J'+2^{M-1}>]. \quad (21)$$

With the same method, we can get the following possible *b*:

$b=[<I', J'+2^M>,<I'+2^{M-1}, J'+2^M+2^{M-1}>]$. (22)
$b=[<I'+2^M, J'>,<I'+2^M+2^{M-1}, J'+2^{M-1}>]$. (23)
$b=[<I'+2^{M-1}, J'>,<I'+2^M, J'+2^{M-1}>]$. (24)
$b=[<I'+2^{M-1}, J'+2^M>,<I'+2^M, J'+2^M+2^{M-1}>]$. (25)
$b=[<I'-2^{M-1}, J'>,<I', J'+2^{M-1}>]$. (26)
$b=[<I', J'+2^{M-1}>,<I'+2^{M-1}, J'+2^M>]$. (27)
$b=[<I', J'-2^{M-1}>,<I'+2^{M-1}, J'>]$. (28)
$b=[<I'+2^M, J'+2^{M-1}>,<I'+2^M+2^{M-1}, J'+2^M>]$. (29)
$b=[<I'+2^{M-1}, J'+2^{M-1}>,<I'+2^M, J'+2^M>]$. (30)
$b=[<I'-2^{M-1}, J'+2^{M-1}>,<I', J'+2^M>]$. (31)
$b=[<I'+2^{M-1}, J'-2^{M-1}>,<I'+2^M, J'>]$. (32)

All possible *b* indicated by (21)~(32) is shown in figure 7. In this figure, the *d2*-value of the lower-resolution block ABCD will be affected by those of blocks (21)~(32). This figure has the same meaning as figure 5.□

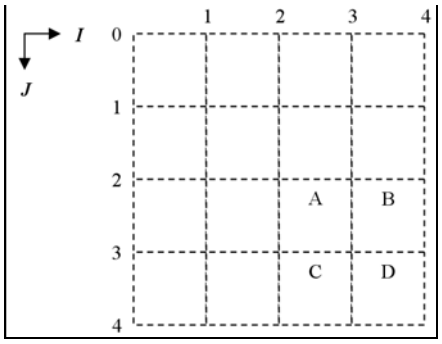

Figure 6: The Coordinate System

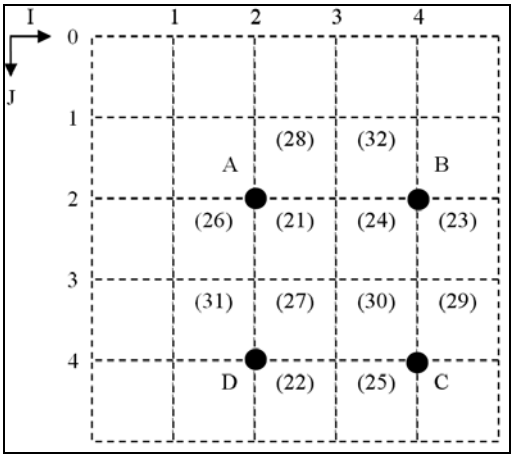

Figure 7: All Possible Higher-resolution Blocks Which *d2*-values Put on an Effect on a Specific Lower-resolution Block

Following this theorem, we can use the algorithm above to render dynamic terrain such as a terrain where groups of bombs make craters in a single tile. When the dynamic terrain crosses tiles, we should use an enhanced version of this algorithm in section 4.

3.2. Image/Texture Updating

When terrain changes dynamically, textures or images used for rendering the terrain usually need to be changed. However, we cannot change the images directly due to the fact that libMini employs S3TC (Brown 2009) algorithm to compress textures. S3TC compresses every block of 4×4 RGB or RGBA pixels into 64-bit data. Therefore, we can take four steps to update textures.

First, we calculate the image scope of the dynamically changed terrain and align it to make sure that both width and height of it are multiples of 4 and get the highest-resolution image data in this aligned scope from libMini. The second step is to decompress the aligned image and then to modify some decompressed image pixels according to the dynamical terrain model. The following step is to recompress the modified data into the form which libMini can recognize. At last, we use the modified highest resolution image to create images in other lower resolutions and reload the revised image data into memory according to the current run-time LOD.

If an image to be changed crosses n tiles (n is 1, 2 or 4), as shown in figure 8 (Yang 2009), following the 4 steps above, we need 9 steps to update the terrain image: calculate the affected areas of all tiles, align the areas, collect highest-resolution image data from all tiles, decompress them, combine the decompressed data into one image then modify the whole image according to terrain model, divide the whole image into n parts, compress each part, recalculate lower-resolution images, and in the end, rerender the images.

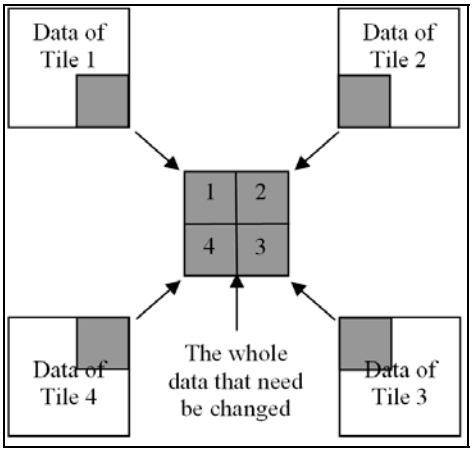

Figure 8: Processing Images Crossing Tiles

4. BLENDING BETWEEN TILES

The libMini guarantees that the difference of roughness level between two adjacent blocks within one terrain tile is less than or equal to one. It doesn't guarantee such a level difference between tiles especially when the adjacent titles change independently. As a result, there may produce holes between tiles especially when the terrain dynamically changes in the margin of tiles. The number of the influenced tiles will be 1, 2 or 4 when the terrain changes. These tiles are called target tiles. To avoid holes between target tiles, we take a 3-step blending method.

First, based on a logic coordinate system we update elevation of changed terrain to make sure that the joint elevation values of the two tiles be the same. As shown in figure 9 (Yang 2009), two lines of points covered in one ellipse are managed by two tiles. In fact, the two lines are the same. When one tile is updated on points of the same line, they may be also changed by update to the other tile. So there are two copies of elevation of the same line. In the following process of *d2* calculating the resolution levels of grids near the same line probably produce a difference greater than 1 to make holes. To keep these points the same elevation value, we treat the terrain being changed as a whole part in separate logic coordinates. After the terrain in such coordinates is updated, the elevation values of the terrain are mapped back into every tile in their coordinates.

Second, all target tiles are processed as if they were one whole tile in the process to calculate *d2*. The bottom-up algorithm in section 3.1 is used here to reduce the amount of terrain blocks to be updated. Therefore, a difference not greater than 1 of resolution level between tiles can be guaranteed and possible holes among tiles can also be eliminated.

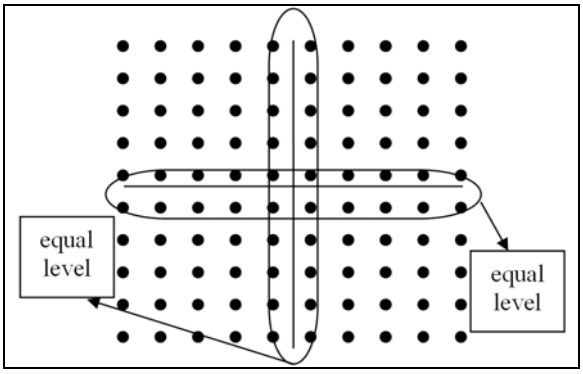

Figure 9: Keep Joint Elevation Values the Same

5. ALGORITHM PERFORMANCE AND EXPERIMENT RESULT

If the size of a terrain tile is $2^N \times 2^N$, time complexity of top-down algorithm in libMini is shown as (33) where C_1 is time for calculating *d2* and C_2 is time for passing *d2* to 3 lower-resolution blocks. We suppose that the scope of elevation data being changed is $2^n \times 2^n$, time complexity of botttom-up algorithm is shown as (34) where C_3 indicate time for indexing *d2* value of 12 higher-resolution blocks. Generally C_3 of is 4 times greater than c2. But usually *n* is about 3 and *N* is greater than 10. In conclusion, bottom-up algorithm is about 10^3 times faster than the old one and is more adaptive to situations such as war games where local changes to elevation are numerous and frequent (about 10^3/second).

$$(C_1+C_2)\sum_{k=1}^{N} 2^{2k} \approx \frac{4(C_1+C_2)}{3} 2^{2N} \tag{33}$$

$$(C_1+C_3)\sum_{k=1}^{n} 2^{2k} \approx \frac{4(C_1+C_3)}{3} 2^{2n} \tag{34}$$

We test the algorithms above on a PC with Pentium 2.8GHz CPU and 2GB memory. The programming environment is Visual Studio C++ 2005. The tested terrain consist of 16×16 tiles and each tile has 257×257 grid points and 2048×2048 pixels and dynamic terrain is created by a crater model covering about 16×16 grids. The average time to render the dynamic terrain for 1000 tests is about 1.27 milliseconds whereas the algorithm of libMini takes is 2512 milliseconds. The process of test is described below.

Firstly we build a type of crater model. The parameters of the model are defined by a quadtuple <position, direction, radius, depth> where position indicates the position of bomb explosion (the center of a crater), direction is the direction of the bomb track, and radius and depth are the radius and depth of the crater respectively.

The intersection line that a longitudinal section intersects the crater elevation plane is shown in figure

10 (Yang 2009). The numbers on this figure represent distance from the center point of the crater. If the radius of the crater is *r*, then the number 2 has the meaning of 2*r*.

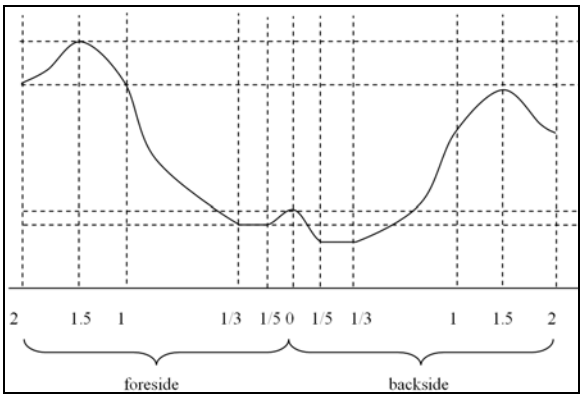

Figure 10 The Intersection Line of the Crater

Figure 11 shows how the elevation of a cross-4-tile crater is updated. To be clearer, we mark the joint part of the tiles red and the crater green. Figure 12 presents the updated crater image. Also to be clearer, one tile image left blank and the crater is lightened and marked mainly khaki.

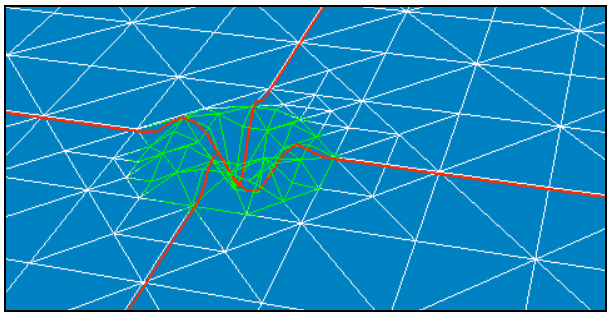
Figure 11: Update on Elevation of a Cross-4-tile Crater


Figure 12: Update on Image of a Cross-4-tile Crater

ACKNOWLEDGMENTS

The authors wish to thank associate professor Zhang Jinfang who developed a software product based on both VTP and libMini and to thank Yang Kai (Yang 2009) who implemented the algorithm of this paper.

REFERENCES

Lindstrom P., Koller D., Ribarsky W., Larry F.H., Faust N., Turner G., 1996. Real-time Continuous Level of Detail Rendering of Height Fields. *Proceedings of the 23rd Annual Conference on Computer Graphics*, pp. 109–118. August 4-9, New Orleans (LA, USA).

Hoppe H., 1996. Progressive meshes. *Proceedings of the 23rd Annual Conference on Computer Graphics*, pp. 99–108. August 4-9, New Orleans (LA, USA).

Duchaineau M., Wolinsky M., Sigeti D.E., et al, 1997. Roaming Terrain: Real-time Optimally Adapting Meshes. *Proceedings. of the 8th Conference on Visualization*, pp. 81-88. October 18-24, Phoenix (Arizona, USA).

Jin H., Lu X., Liu H., 2009. View-dependent fast real-time generating algorithm for large-scale terrain. *Proceedings of the 6th International Conference on Mining Science & Technology*, pp. 1147-1151. October, 18-20, Xu Zhou (Anhui, China).

Robert W., James F., Jessica K., 1999. Animating Sand, Mud, and Snow. *Computer Graphics Forum*, 18(1), 0-11.

Shamir A., Valerio P., Chandrajit B., 2000. Multi-resolution Dynamic Meshes with Arbitrary Deformations. *Proceedings of IEEE Visualization*, pp. 423-430. October 8-13, Salt Lake City (Utah, USA).

He Y., Cremer J., Papelis Y., 2002. Real-time Extendible-resolution Display of On-line Dynamic Terrain. *Proceedings of Graphics Interface*, pp. 27-39. May 27-29, Calgary (Alberta, Canada).

Cai X., Li F., Sun H., Zhan S., 2006. Research of Dynamic Terrain in Complex Battlefield Environments. *Lecture Notes in Computer Science*, 3942:903-912.

Shiben B., Suryakant P., Narayanan P.J., 2008. Real-time Rendering and Manipulation of Large Terrains. *Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, pp. 551-559. December 16-19, Bhubaneswar (India).

Stefan R., Wolfgang H., Philipp S., Seidel H.P., 1998. Real-Time Generation of Continuous Levels of Detail for Height Fields. *Proceedings of WSCG*, pp. 315-322. February 9-13, Plzen (Czech Republic).

Stefan R., 2009. *Real-time Terrain Rendering*. Available from: http://stereofx.org/terrain.html [accessed 4 July 2010].

Discoe B., 2009. *Virtual Terrain Project*. Available from: http://vterrain.org [accessed 4 July 2010].

Discoe B., 2005. Open-Source Visualization and the Virtual Terrain Project. *Geo: Connexion International magazine*, pp. 47-50.

Brown P., 2009. *EXT_texture_compression_S3TC*. Available from: http://www.opengl.org/registry/specs/EXT/texture_compression_s3tc.txt [accessed 4 July 2010].

Yang K., 2009. *Research on Visualization of Dynamic Terrain (in Chinese)*. Thesis (Master Degree). Institute of Software, Chinese Academy of Sciences.