Efficient Rendering Large Terrains using Multiresolution Modelling and Image Processing Techniques

Ömer Nebil Yaveroğlu

Abstract—In this study, we propose an algorithm for rendering large scale terrains given as height field files. It is costly to render terrains in a computer graphics applications. In the basic implementation, the vertices given by the height field file should be combined by triangles. A height field file of size 1024x1024 pixels can generate up to 2 million triangles when the whole given data is rendered this way. Even with today's technology, it is not possible to render this amount of data efficiently.

To overcome this problem, the simplest solution is to reduce the number of triangles needed to render the scene in such a way that the rendered image quality is not effected. There are several techniques applied under the name of multiresolution modelling, which aims to reduce the number of triangles to be rendered by generating several different levels of details. We have developed a multiresolution rendering technique in this paper, which uses image processing techniques to decide on the detail levels. With the application of the method we developed, it is possible to reduce the number of triangles to be rendered upto 49%.

Keywords—Terrain Rendering, Median Filtering, Edge Detection, Multiresolution Modelling

I. Introduction

Rendering of height fields is a well-known problem in computer graphics. For the generation of the terrains in games or simulations, thousands of polygons have to be drawn repeatedly in real time. Even with the current improvements in graphics hardware, it is still a difficult problem to render terrains with changing height values. Many studies have been performed for solving this problem. Some of these studies aim to reduce the complexity by applying approximations on the polygons to be drawn reducing the image quality. Some other studies apply clipping on the whole terrain depending on the camera position in order to reduce the number of polygons to be processed. All these approaches come with some pros and cons.

In the *Literature Survey* part of this paper, we will shortly describe several solutions proposed to the problem. In the *Methods and Results* part of the paper, we will describe the solution developed by us in detail. You can find a short summary of this paper in *Conclusions* part. We finalize the paper mentioning the *Future work* required to improve the performance results of the study.

II. Literature Survey

In this part of the paper, we would like to mention the main approaches and problems of the solutions for the rendering of height fields. We will also focus on the solutions offered by two papers which affected our solution to the problem most ([1],[2]) .

The main aim of the algorithms proposed as a solution to terrain rendering is rendering the terrain in most realistic and visually rich way with the minimum computational and hardware requirements. For this aim, nearly all of the algorithms use different Level of Detail (LOD) algorithms in order to reduce the number of polygons to visualize the terrain. For the adaptive modeling of terrains, a quad tree data structure is used to keep the height field data. The height fields are provided to the program as two dimensional matrices. The simplest way to visualize the height values provided with these matrices is combining them by quads. A quad-tree representation of the data allows multi-resolution modeling of the data in a cost effective way. In order to reduce the rendering complexity, the quads in the quad tree can also be rendered as triangle fans. Current graphics hardware has efficient rendering properties for triangle fans. It is possible to render larger terrains by using triangle fans than it is possible to render by rendering with quads. Usage of triangle fans also reduce amount of error and produce visually satisfactory terrains.

One way to reduce the number of polygons to be drawn is grouping triangles having similar values into larger triangles. With this approach, the smoother regions in the terrain are combined and more detail is included for the regions that are rough. Although usage of similar height values to group the polygons together forming larger polygons reduce the number of polygons to be drawn, the number of polygons can further be reduced by using the distance to the camera position. In real life, we see the details of the objects which are closer to us more than the objects that are far away. Same thing applies on terrains. Even though we see the smallest changes on the terrain surface which are close to us, we do not recognize that much detail if the terrain region we are looking is far away. We just see a silhouette of the largest heights but not the details. This fact can be used to reduce the number of polygons while rendering the environment. The objects that are far away from the camera can be rendered with more tolerance to errors. This change will create larger polygons for regions that are far away from the camera position.

Even though the idea of changing the detail level in order to reduce the number of polygons to be rendered is simple and effective, it comes with a number of problems. One basic problem is deciding on the level of detail to be used in different regions of the height field. Most of the methods use different

Ö. N. Yaveroğlu is with the Department of Computer Engineering, Middle East Technical University, Ankara, 06531 TURKEY e-mail:nebil@ceng.metu.edu.tr

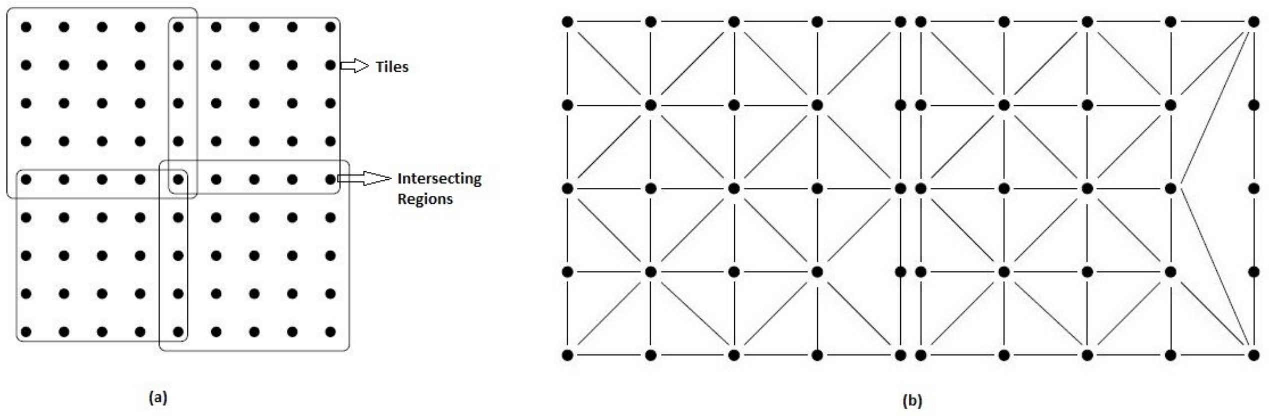


Fig. 1. The image (a) represents the tiles and their intersection. Image (b) shows how cracks are avoided by rearranging the triangles forming the crack. The image on the left shows the case that can cause cracks. The image on the right shows the rearrangement to avoid the cracks.

error metrics to make the decision of the level of detail. The distance to the camera and surface roughness become parameters of these approaches. Another problem is cracks which occur at the borders of different levels of detail. The reason for crack occurrence is the mismatching of the height approximations with the real height values. In order to get over this problem, many different approaches have been used such as crack filling, mapping these regions on the lower resolution sample, adding T-vertices to higher resolutions and changing connectivity of vertices for the higher level of detail polygons.

For the above problems, the camera position is static. Even more problems occur when the camera location can be changed. A problem called popping occurs while changing the level of detail. Sudden changes in level of detail cause this problem. Nearly all implementations of terrain rendering allow a change of just one level between neighbor regions to minimize this problem. But even with this precaution, popping can be observed as the camera position changes. The basic solution to this problem is performing an animation called morphing while changing the level of detail. Morphing is performed by adapting the positions of the points to the new resolution in a time period.

Apart from these main algorithm dependent issues, there are several problems that need to be studied. Texturing the terrain with this multiple resolution values [4], performing the rendering processes on just GPU without any computational overhead to CPU [6], performing the calculation process on parallel computers are some of the current research challenges. But in the scope of this study, we will not be interested in these problems since they are out of the scope of our project.

After mentioning the main problems of terrain rendering and the solutions to these problems, we would like to provide the solutions provided by *Ulrich et al.* [1] and *Larsen et al.* [2]. In the study of *Ulrich et al.*[1], a solution for rendering large, static out-of-core datasets is proposed. The paper includes a general overview of the problems and solutions in terrain rendering. They use a Chucked LOD algorithm for generating the detail levels to be used during the rendering process. At the heart of the method, a static tree is constructed with largely independent preprocessed meshes at the preprocessing step. The meshes keep the primitives that can be drawn directly in a single rendering call. As you proceed up on the tree, you get into the lower resolutions of the same object. As a matter of fact, this approach is followed in many of the other studies. Quad-tree representation is a special data structure which allows the use of chunked LOD algorithms. After the construction of the quad-tree at the preprocessing step, they compute maximum screen space vertex error for view dependent rendering. With this error metric, they decide on the level to render the quad. After deciding on the LOD to represent the height field quads, problems such as cracking and popping need to be eliminated. They overcome the cracking problem by applying a crack filling algorithm. Crack filling is performed by introducing new polygons at the borders of different levels of details to fill the gap. For avoiding the popping problem, they use simple morphing. They simply animate the adaptation of the vertices to new locations by slightly moving the location at each step. The study [1] is successful except their solution to the cracking problem. With a slight modification, this method would be a lot more effective.

In another study performed by *Larsen et al.* [2], the terrain is divided into equal sized tiles. The sizes of the tiles are reduced when more detail is required. Dividing the tiles as needed, they perform a similar algorithm to Chunked LOD. The difference from the application of Chucked LOD algorithm in *Ulrich et al.*'s study [1] is that they allow the tiles to intersect. These intersecting areas form some kind of transaction regions which become helpful in crack elimination. They eliminate cracks by rearranging the triangles forming these intersecting regions. An illustration of these regions and crack elimination is given in Figure 1.Also their solution to popping is the application of morphing. We will not provide the details of morphing since we do not propose a solution for the popping problem.

There are some important proofs in *Larsen et al.*'s paper [2]. One of these proofs is that usage of display lists results with better performance when compared to direct usage of polygons. In a similar way, usage of connected drawing primitives such as strips and fans result with better performance when compared to direct usage of polygons but worse than display lists. This is a result of the working mechanisms of graphics cards. The disadvantage of using display lists is that it is not possible to modify the geometry after the display list is created. Another proof that they perform is that only 5 levels of detail is enough to reduce the complexity. When the tiles are divided into smaller tiles of four, 99.61% reduction percentage can be achieved at the fifth detail level showing that it is not meaningful to use any more levels of detail since they have no effect on the polygon number reduction. It was important to keep these implementation tricks in mind while making the implementation.

The main bottleneck in the current implementations of multiresolution modelling is the high computation requirements in order to decide on the level of detail. Also crack filling and morphing includes some more computational cost on CPU. So the rendering and computation processes should be carried to GPU as much as possible.Also by this way, it would be possible to perform the computations in parallel.Because of that recent researches are focused on these aspects of the terrain rendering problem. Studies are being done on texturing the terrain or rendering terrain on just GPU with parallel processing which are out of the scope of our project.

III. Methods and Results

In our implementation, we have used the quad tree representation of the height fields in order to use the Chucked LOD algorithms. Using a top-down approach, we have decided on the level of detail for different quads forming the terrain. We decided the level of detail to render a quad by using an error function which uses the distance to camera position and surface roughness as parameters.

We have directly computed the distance to the camera position. Depending on the distance to the camera position, we have decided on the tolerance amount for the error occuring as a result of approximation. By finding a constant factor, defining the tolerance to the error, we allowed further away tiles to be rendered in lower resolution. Similarly, this constant factor resulted in more detailed tiles for the locations close to the camera position.

In order to decide depending on the surface roughness, we have used an image processing technique called median filtering. Median filtering is an image smoothing algorithm. By applying this smoothing algorithm, we have averaged the values in the height field removing the sudden changes in the height value. By comparing the error between the smoothed height field and original height field, we have found the regions where the surface has sharp differences. In other words, by smoothing the heightmap file and then taking the difference from the original image, we have found the reigions that are rough. In these rough regions, we rendered the terrain in high resolutions and vice versa.

To avoid cracks, we have rearranged the triangles of the lower resolution tiles at the boundary regions. While removing the cracks, this method produces more realistic looking images since it increases the resolution just at the borders of the resolution changes. We have not implemented a method for the popping problem. Since our solution resulted with low FPS values, we could not observe popping clearly. For avoiding any unnecessary computational overhead, we did not take the popping problem into account in our implementation. The one and only solution to popping is applying morphing and it is quite straight forward to implement. The morphing can easily be included into our solution. This is why our implementation does not provide a solution to popping for the moment.

We have compared the performance of our method by counting the number of polygons rendered and by the FPS value produced during the rendering process. In the following subsections, we provide the details of our implementation and the results of this applied method.

A. Construction of Quad Tree

For the methods to be applicable on the height field data, an efficient data structure should be constructed. For this purpose, quad tree's are found to be the most suitable data structure. In most of the research done in the field, quad trees is the most popular data structure and it is used in nearly all of the studies. This data structure fits the nature of the problem. Since the tiles are considered as quads and since it is easy to divide the tree symetrically as much as wanted, quad trees are the best choice for keeping the height field data.

For the experiments we have made, we have used height field data with a maximum size of 1024x1024 pixels. The main reason for this assumption is that 2048x2048 height field files does not fit into the memory of our computer. The highest resolution we could use for height field data was 1024x1024 so we have developed our implementation accordingly.

During the construction of the quad tree, we have taken the maximum size to be the root of the quad tree. We have divided this tile into four generating the child nodes. Then we have again divided the child nodes into four. This process goes on for each generated child independently until the quad size of the child is at the size of maximum resolution or until the error metric is found to be small enough meaning that there is no need for any more divisions. The computation details of this error metric is described in the *Deciding on the Level of Detail (LOD)* part of this paper.

B. Deciding on the Level of Detail (LOD)

One of the most important contribution we make to the problem of terrain rendering is the Level Of Detail(LOD) decision mechanism we have used. While deciding the Level Of Detail to render a tile of the terrain, we have considered the distance to the camera position and the surface roughness.

In order to decide on whether to generate more children from a node or not, we compute an error metric. For each node, this error metric is computed and the node is divided depending on this computed value. The following formula is used in order to compute this error metric.

$$errorMetric = \frac{\text{Average Error * Camera Constant}}{\text{Average Quad Error}}$$

In this equation, *Average Error* and *Average Quad Error* are parameters computed for measuring the surface roughness.

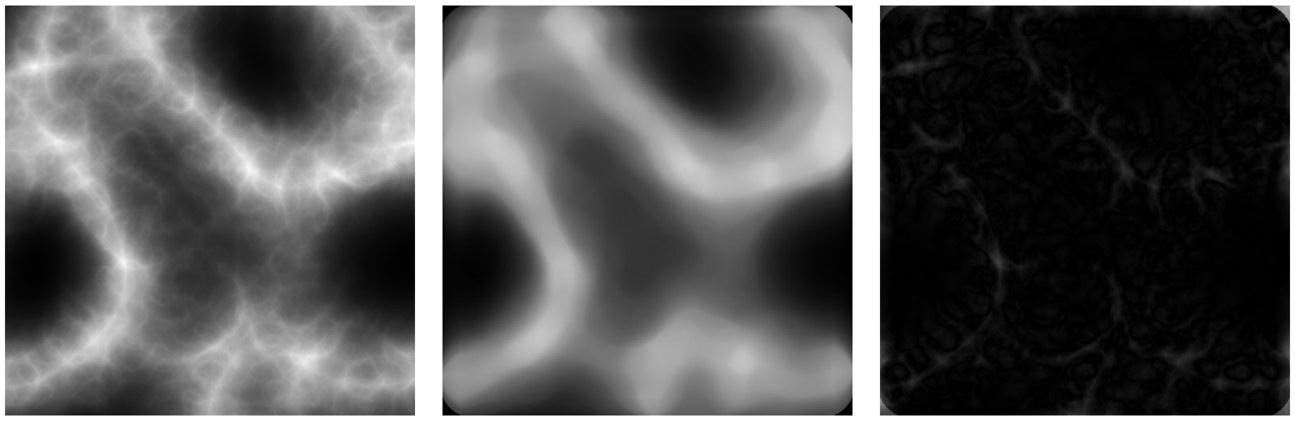

(a) Original Image (b) Smoothed Image (c) Difference Image

Fig. 2. The steps of processing the height field file using image processing

Camera Constant is computed depending on the camera position and quad position.

As mentioned earlier, an image smoothing technique is used to find the surface roughness. The given height map is smoothed and the difference from the original image is found by comparing the smoothed image with the original image. The details of this process will be given in *Application of Image Processing Techniques* part. But for the moment, in order to describe the computation of *Average Error* and *Average Quad Error* parameters of the above formula, we will use the difference image term in order to mention the data found by comparing the original image and the smoothed image. *Average Error* is the average of the error values in the difference image for the quad represented by the considered node. For the quad represented by the considered node, the corresponding values of the difference matrix are summed and divided to the number of items summed in order to get the average error value of the quad. *Average Quad Error* is nothing but the *Average Error* of the root of the quad tree. It is computed once at the beginning of the quad tree construction and then used as a decision parameter for the surface roughnes.

The *Camera Constant* parameter is computed with the below formula.

$$CameraConstant = \frac{(\text{Xcam - Xnode})^2 + (\text{Zcam - Znode})^2}{\text{Distance Metric}^2}$$

In the above equation Xcam and Zcam parameters are x and z coordinates of the camera. Similarly Xnode and Znode parameters are the x and z coordinates of the center of the tile to be drawn. *Distance Metric* parameters is a constant distance. It can be though as the distance that is used to lower the resolution one level down. It does not work exactly like this but the intuition behind is the same.

If *errorMetric* is less than or equal to 1, the quad is divided into four children quads. Else the error caused by the quad can be tolerated so the quad is not divided any further.

C. Application of Image Processing Techniques

As mentioned in the previous parts, median filtering is used in order to find the rough regions in the given height field file. The median filter is a non-linear image processing technique, often used to remove noise from images or other irregularities from the image files. The main idea of median filtering is based on calculating the median of neighboring pixel values. After deciding on the size of the filter to be used, this filter is passed over the whole image file. At each step of this iteration, the values under the filter are sorted and the median of these values is taken to represent the value at the center [7].

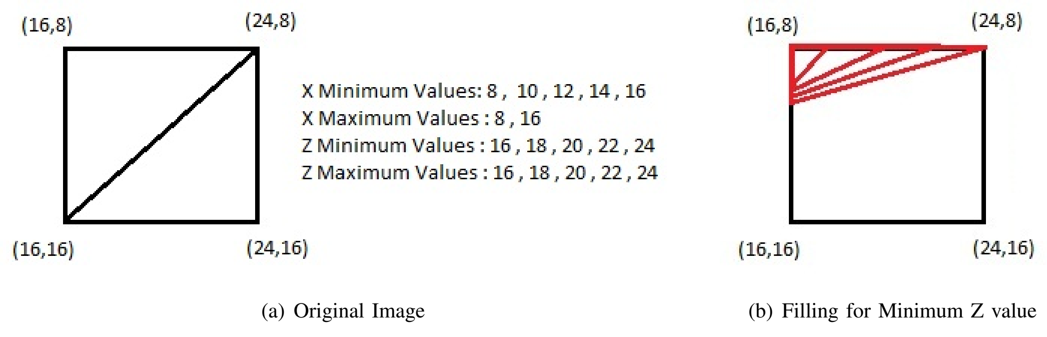
We have used median filtering to smooth our height field images. By smoothing our image, we eliminate the sharp difference changes. We have used a median filter of size 16x16. The decision on the size of the median filter is made depending on the quad size of highest resolution node. Also a visual comparison is made in order to determine whether the selected filter size is appropriate or not.

After applying median filter on our heightmap file using MATLAB, we have compared the original height field file and the smoothed height field file. For each pixel of these two images, we have substracted the original height value from the smoothed height value. This process gives us the regions that have sharp changes in the height value. A sample height field file, smoothed version of it and the difference image are given as Figure 2.

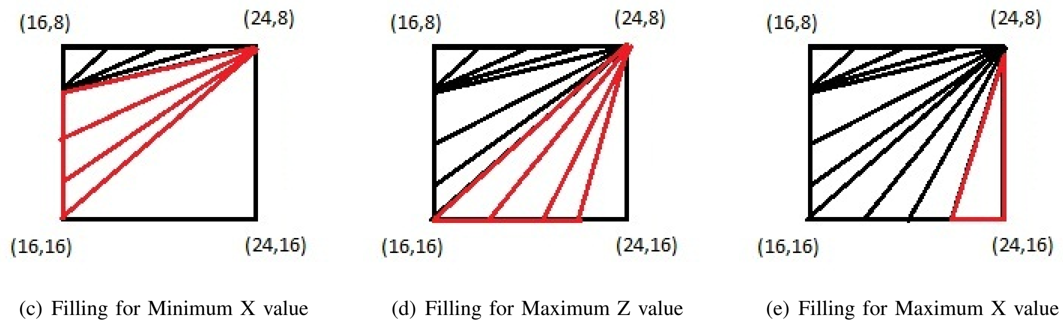
D. Correction of Cracks

Several crack elimination and prevention methods are explained previously in the *Literature Survey* part of this paper.Among them we have chosen to use a crack prevention technique in order to not to increse the polygon count while generating a non-realistic image. For this reason, we have chosen to apply a reordering on the triangles to render a tile at the borders of resolution changes. For an intersection of high resolution and low resolution tile, we prefer to rearrange the triangles of the lower resolution tile. In other words, we divide the bigger tiles in order to match their height values with the corners of the lower resolution tiles.

For crack prevention, prior to rendering the tiles, we first have to find the regions that cracks occur. We have implemented a recursive algorithm which searches neighbors of a node that can cause cracks recursively. Our recursive algorithm has a top down approach. So it starts from the root of the quad


(16,8)
(24,8)
X Minimum Values: 8 , 10 , 12 , 14 , 16
X Maximum Values : 8 , 16
Z Minimum Values : 16 , 18 , 20 , 22 , 24
Z Maximum Values : 16 , 18 , 20 , 22 , 24
(16,16)
(24,16)
(16,8)
(24,8)
(16,16)
(24,16)

(a) Original Image (b) Filling for Minimum Z value



(c) Filling for Minimum X value (d) Filling for Maximum Z value (e) Filling for Maximum X value

Fig. 3. The steps of the crack filling process made during rendering for a low resolution tile . For the lists returned for X maximum and X minimum values corresponding z values which can cause cracks are returned as a list. Similarly for Z maximum and Z minimum values corresponding x values are returned.

tree and iterates searching the next child that can be a neighbor of the considered node. After finding the neighbors that the cracks can occur, we generate an ordered list of the cracking points from the corners of the lower resolution tiles. Using this ordered list, we rearrange the triangles of the lower resolution tile.

The rearranging procedure is a bit complicated. We tried to use triangle fans as much as possible since it is proven that it is faster for a graphics card to render a triangle fan compared to individual triangles. In our reordering algorithm, we first eliminated the cracks occuring at the minimum z value of the quad by combining the list returned for the minimum z value. We take the second element of the list returned for minimum x value. Taking this element as the pivot, we generate a triangle fan with the elements in the ordered list returned for minimum z value. Then we take the maximum x and minimum z value of the quad as the pivot point and generate a triangle fan with the ordered list returned for minimum x value. From the second element of the list returned for minimum x direction to the last element we take the vertices eliminating the cracks in minimum x value of the quad. We continue this triangle fan removing the cracks for maximum z value. We include the list of points returned for the maximum z value until the last two elements. When we reach the last two elements of the list generated for maximum z value, we change the pivot element of the triangle fan. We take the element before the last element from the list returned for the maximum z value as the pivot element. This time we iterate on the list returned for maximum x value taking the vertices in the list. When written, the algorithm seems very complicated but the process can be understood better in Figure 3.

E. Results

We have implemented our algorithm using OpenGL with C++ on a laptop with 512 Nvidia Geforce 9500 M graphics card, Intel(R) Core(TM)2 Duo CPU T9300 @ 2.50GHz processor, 2013 MB's of memory. We have performed this implementation starting from nothing. So no previously implemented libraries or implementations have been used for performing this implementation. We have achieved good results by means of reducing polygon count. But since we have made the implementation just of CPU, FPS values are quite low because of the computational overhead of the construction and manipulation of the quad tree.

We have used two different benchmark height fields to test our algorithm. You can see these height field images in Figure 4.We have not chosen these height fields randomly. The first image is a smooth one where there are no sudden changes in the height values. When it is rendered, it produces a terrain consisting of two mountain like heights and a valley. The second image is used for just seeing the performance of the algorithm in a very complex scene.

In Figure 5, you can see the results of three different algorithms we have applied to render the scene. These three algorithms are the direct terrain generation with no optimization, multiresolution terrain generation without crack filling and multiresolution terrain generation with crack filling. While comparing our performances on reducing the number of polygons and increasing FPS values, we have used these three algorithms. During the generation of the figures we have

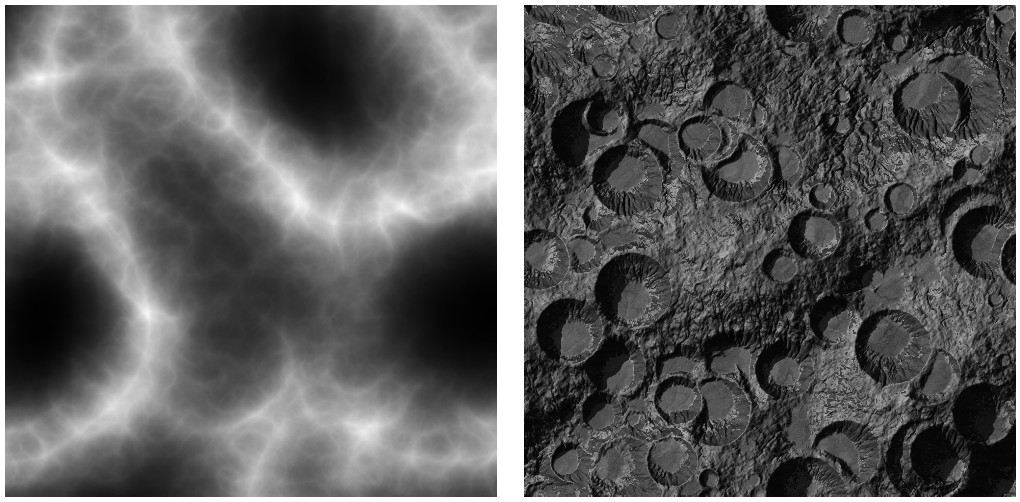

(a) Smooth Height Field (b) Complex Height Field

Fig. 4. The benchmark height fields used for testing purposes

implemented illumination using Flat Shading. There are two reasons for us to choose flat shading in this implementation. The first one is that it is computationally cheaper compared to Gouraud Shading. We didn't want to lose computational time for calculating the average vertex normals. Another reason for us to choose flat shading is that it allows us to see the polygons rendered. By this way, we can recognize which region is rendered in lower resolution and which region is rendered in more detail.

As we have told before, we have applied the three algorithms on the two benchmark height field files. Their performances on the number of triangles they generate and the FPS values that could be achieved are listed in Table I and Table II.As can be seen from the tables, the number of triangles can be reduced 49% for the smooth image and 33% for the complex image. These results were expected. Since the complex image has many height changes, the algorithm does not want to lose the height information. For that reason the tiles are rendered in higher resolution resulting with lower reduction on the number of polygons. But for a smooth image, the results are very promising. It can be seen from Figure 5 that even when the image is rendered in multiresolution with crack filling, the resulting images are quite similar to the original image.

From Tables I and II, it can be seen that for the multiresolution model without crack filling, the FPS values have increased with the reduction of the number of polygons. But for the crack filled case, the results were worse than even direct rendering. The reason for this problem is that, we have implemented crack determination in the display call back. So each time the polygons should be rendered, the cracks will be calculated again and again. Even though it seems like an implementation error, in fact it was a conscious decision at the time of implementation. We have thought that keeping the neighbors of each tile would result with huge memory cost. In order to avoid that we have performed crack finding in display callback function. This resulted with reduced performance by means of FPS. But it can still be a promising solution for high FPS values with a bit of modification.

IV. Conclusions

In this study, we have tried to reduce the number of triangles required to render a terrain without losing significant visual information. For this aim, we have implemented an algorithm using median filtering. We have used median filtering to smooth the given height field image. Using this smoothed and original image, we have determined the regions which have sharp changes of height values. These regions with sharp height changes and regions close to the camera position are rendered in high detail. We have also developed a new crack correction method in our implementation in order to fill the gaps occuring between different levels of details.

We have achieved good results by means of reducing the polygon count. We have reached up to 49% polygon count reduction. This is really a high amount of improvement. By means of FPS value, we again achieved high values when the crack filling algorithm is not activated. The reduction on the number of polygons can even increase the FPS value 2 times. But for the crack filled algorithm, since the positions where the cracks occur are found by searching the quad tree at the display call back, FPS values are reduced in huge amount. If the neighbors were determined outside the display callback, we would again get high FPS values. But this change would require more memory since neighboring information should be kept when they are computed outside the display callback.

V. Future Work

The first improvement to be done is the correction on the crack determination algorithm as we mentioned in the *Results* and *Conclusions* parts of the paper. Since huge computational cost is introduced as a result of the computation of the cracking regions again and again, the FPS values extracted from the crack filling algorithm are quite low. If the neighbors were calculated outside the display callback and kept in the quad tree, the algorithm would perform a lot better.

TABLE I
PERFORMANCE COMPARISON FOR THE THREE ALGORITHMS APPLIED ON THE SMOOTH HEIGHT FIELD FILE

	Direct Rendering		Without Crack Filling		With Crack Filling	
Minimum Tile Size	FPS	Number of Triangles	FPS	Number of Triangles	FPS	Number of Triangles
2x2	3.75	522242	7.57	243260	0.96	267236
4x4	15.06	130050	25.61	70430	3.54	76734
8x8	61.44	32258	90.36	20102	12.19	21647
16x16	246.75	7938	313.06	5672	43.26	6065
32x32	867.13	1922	985.01	1580	161.67	1681

TABLE II
PERFORMANCE COMPARISON FOR THE THREE ALGORITHMS APPLIED ON THE COMPLEX HEIGHT FIELD FILE

	Direct Rendering		Without Crack Filling		With Crack Filling	
Minimum Tile Size	FPS	Number of Triangles	FPS	Number of Triangles	FPS	Number of Triangles
2x2	3.63	522242	5.41	336626	0.73	354323
4x4	15.03	130050	19.15	91094	2.73	95806
8x8	60.96	32258	72.85	24584	10.28	25886
16x16	246.75	7938	265.46	6680	37.43	7003
32x32	876.24	1922	985.01	1808	142.147	1857

There can also be some improvements in the decision mechanism of the level of detail. Some more parameters can be added to the computation to generate more realistic images. Also in order to reduce the number of triangles send to graphics card to be rendered, clipping of the regions that are not visible can also be implemented.

On the other hand, the performance values can be increased in great amounts if the algorithm is parallelized and implemented by GPU programming. This change would reduce the computations done by CPU which is the current bottleneck of the implementation. The reduction percentage on the number of polygons is still promising and the method can be used more efficiently when GPU programming is included.

REFERENCES

[1] Ulrich, T. (2002). Rendering Massive Terrains using Chunked Level of Detail Control.

[2] Larsen, B. D.; Christensen, N. J. (2003). Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail. Journal of WSCG, Vol.11, No.1, ISSN 1213-6972. Plzen, Czech Republic: UNION Agency Science Press.

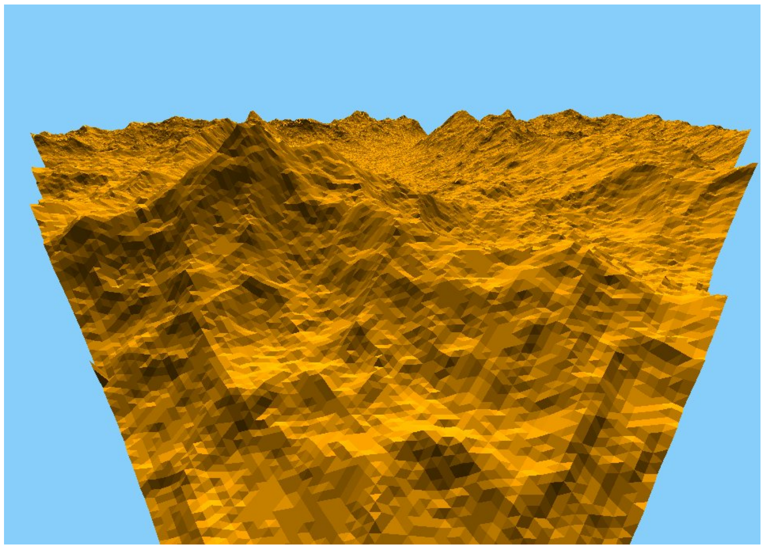
[3] Boer, W. H. (2000). Fast Terrain Rendering Using Geometrical MipMapping.

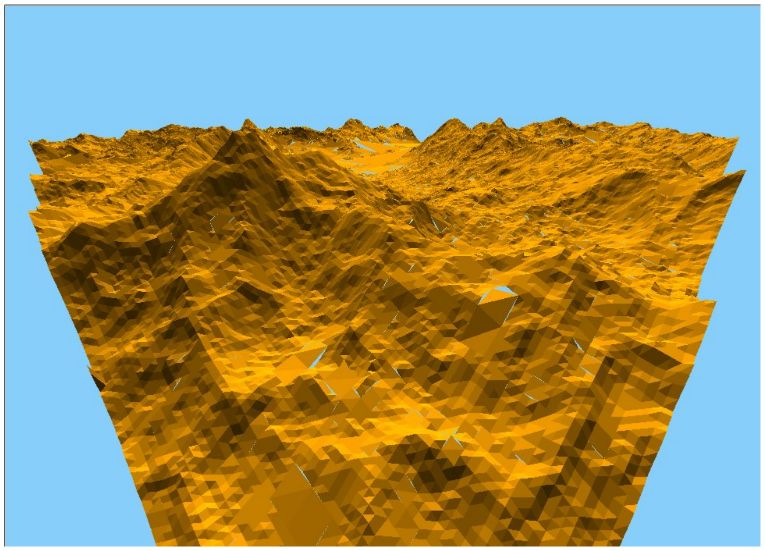
[4] Martin Schneider, R. K. (2007). Efficient and Accurate Rendering of Vector Data on Virtual Landscapes. Journal of WSCG, 2007 .

[5] Stefan Rttger, W. H.-P. (1998). Real-Time Generation of Continuous Levels of Detail for Height Fields. WSCG'98.

[6] Szymon Rusinkiewicz, M. L. (2000). QSplat: A Multiresolution Point Rendering System for Large Meshes. SIGGRAPH 2000. New Orleans, LA, USA.

[7] Lin Yin, Ruikang Yang,Gabbouj, M.,Neuvo, Y.,(1996).Weighted median filters: a tutorial. Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on.Volume: 43, Issue:3,Page(s): 157-192,ISSN: 1057-7130.


(a) Result of Terrain Generation with No Optimization


(b) Result of Multiresolution Terrain Generation without Crack Filling

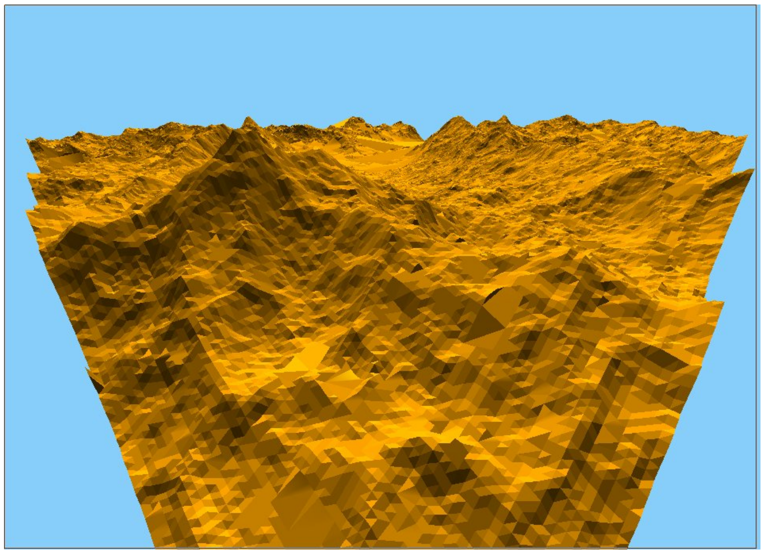

(c) Result of Multiresolution Terrain Generation with Crack Filling

Fig. 5. Results of the application of the three terrain rendering algorithms on the Smooth Height Field file with a minimum tile size of 4x4