PREFACE

First, I would like to thank to my family for supporting and encouraging me throughout my life. I am further indepted to my advisor Prof. Dr. Muhittin Gökmen for guiding me during my masters education and giving me this opportunity to prepare my thesis. I would like to thank Mevlüt Dinç, Erdal Yılmaz and all Sobee Team for sharing their knowledge and data with me. I am also grateful to my office friends, Aydın Karaman, Hakan Demirel, Tuba Şirin, Serap Kırbız and Tolga Esat Özkurt for their friendship and amity.

January 2005 Rupen Melkisetoğlu

CONTENTS

ABBRIVIATIONS

VR : Virtual Reality

GIS : Global Information System

LOD : Level of Detail

FPS : Frames per Second

DLOD : Discrete Level of Detail

CLOD : Continuous Level of Detail

GPU : Graphical Processing Unit

DAG : Directed Acyclic Graph

DDS : Direct Draw Surface

HT : Hyper Threading

TABLE LIST

FIGURE LIST

SYMBOL LIST

f : Subdivision criterion

l : Distance from eye point

d : Node edge length

C : Minimum global resolution

c : Desired global resolution

dh_i : Elevation differences

C_N : View angle dependent minimum global resolution

W : Visible scene window

YÜKSEK ÇÖZÜNÜRLÜKTEKİ ARAZİ VERİLERİ İÇİN DÖRTLÜ AĞAÇ TABANLI 3B ÇİZİM VE SAYFALAMA ALGORİTMASI

ÖZET

Arazi verilerinin görüntülenmesi birçok üç boyutlu uygulamada sıkça karşılaşılan bir konudur. Özellikle uçuş simülatörleri ve ya küresel bilgi sistemleri gibi özellikle birçok askeri uygulamalar için kullanılan bu görüntüleme yöntemi, yükseklik verisinin belli başlı algoritmalarla üç boyutlu hale dönüştürülmesi ile oluşturulur. Varolan bilgisayar donanım teknolosijinin getirdiği kısıtlamalardan dolayı oluşturulan bu üç boyutlu görüntü, basit üçgenlemelerle yapıldığında gerçek zamanlı performans büyük ölçüde düşer. Bu nedenle bakış poziyonuna bağılı olarak yapılan üçgenlemeler, hem oluşturulan üçgen sayısını azaltır hem de gerçek zamanlı uygulamalarda performans kazancı sağlar.

Yüksek çözünürlükteki arazi verileri üzerinde dolaşırken bütün bir arazi verisi bilgisayar belleğine taşınamayacağından, bu veri sayfalama algoritmaları kullanılarak yönetilir. Arazi verilerinin çiziminde kullanılan bu sayfalama algoritmaları hem istenen ölçüdeki araziyi göstermeli, hem de görüntü kalitesini fazla bozmamalıdır, ayrıca uçuş esnasında arka plandan verileri sorunsuzca yükleyebilecek bir yapıya sahip olmalıdır. Üçgenleme dışında arazi verilerinin bilgi içeren bazı bölgeleri detay seviyesi yüksek görüntü verileriyle de kaplanabilir. Bu verilerin arazi verisine eşleşmesi de değişik yöntemlerle yapılabilir.

Bu tez çalışmasında hem arazi verilerinin görüntülenmesi, hem de bu verilerin sayfalanması için iki tane dörtlü ağaç tabanlı algoritma önerilmiştir. Aynı sayfalama algoritması yüksek detaylı görüntülerin arazi verileriyle eşleşmesi için de kullanılmıştır. Bu algoritmalar kullanılarak yapılan bir arazi görüntüleme motoruyla Türkiye'nin üç boyutlu yükseklik haritası çıkarılmış ve hem kaba görüntü dokusu, hem de gerekli bölgelerde detay görüntü dokusu ile kaplanmıştır. Sonuç olarak kalitesi gerçek zamanlı uygulamalar için uygun, esnek bir görüntüleme motoru oluşturulmuştur.

QUADTREE BASED 3D RENDERING AND PAGING ALGORITHM FOR VERY LARGE SCALED TERRAINS

SUMMARY

Visualization of terrain data is a common subject in most of the three dimensional applications. This visualization method covers a large area especially in military applications, such as flight simulators or Global Information Systems (GIS), in which accurate heightfield data is triangulated in such a manner where the result is the three dimensional view of the terrain. Due to the restriction of current hardware technology, creating terrain with simple triangulations reduces the real time performance greatly. Trangulations created according to the eye position of the viewer, reduces the triangle count and increases the real time performance of the simulation.

While navigating through the terrain, it is not possible to get the complete data into computer memory, so this data is managed with paging algorithms. These paging algorithms used in rendering of terrain data should both show the desired scaled area and not reduce the visual quality, furthermore it should have a proper structure to load the data in background without any problem. Other than triangulation, some parts of the terrain data may wanted to be textured with high detail images. Mapping these images on terrain can be done with various methods.

In this thesis, two quadtree based algorithms are proposed for both triangulation and paging of the terrain data. The same paging algorithm is also used for mapping of high detail textures on terrain. A terrain visualization engine is developed using these algorithms and the three dimensional height data of Turkey is created. This data is covered with a rough texture and some necessary parts are covered with detail textures. As a result, proper and flexible terrain visualization engine is created for real time applications.

1 INTRODUCTION

1.1 Overview

Simulation is one of the most important research areas in computer graphics. With simulation we are able to analyze or see things that would not be possible to do in real world, or its cost would be too expensive. Even if we can do things without simulation, analyzing their results in various conditions would consume our time. Simulation is a very wide concept. Due to the development of incredibly high speed computers simulation solutions became more accurate and fast. By this way more difficult or time consuming simulations came to be done easily.

Simulation mimics reality, i.e. everything seems and works like in reality except that it is not real, thus why we call the result Virtual Reality (VR). Computer games are one of the leading industries in virtual reality and many parts of virtual reality solutions can be seen in games. Virtual reality is not only the visualization of these unreal worlds but also behavior of these objects, their dynamics and interaction. But certainly visualization is the most vital part of it. Especially due to computer hardware restrictions many different kinds of problems and solutions exists in virtual systems. Visualization of large scaled and detailed terrains is one of these problems. Rendering these large scaled terrains in computers need to be done with various algorithms in order to achieve the real time performance for human interaction and reduce the hardware technology restrictions.

Terrain rendering applications cover a large area especially in military applications, such as flight simulators or Global Information Systems (GIS), in which accurate heightfield data is triangulated in such a manner where the result is comparable to what one would see in reality [9]. Here most important point is consistent representation of massive terrain surface and its efficient management in computer system, where one can achieve this with a proper paging algorithm.

We must also take advantage of human visual system restrictions in order to decrease the work load in simulation. Simulation is an approximation in many cases, and we can approximate the simulation solutions according to our visual sense quality. The degree of these approximations changes between problems. In highly accuracy dependent problems require more sensitive calculations and more difficult performance solutions. Especially many of the military simulation systems need to be done sensitively and require increased real time performance. So we have to create an appropriate balance within these restrictions and advantages in order to have a good working simulation system.

1.2 Terrain Simulation

What we call terrain here is a two dimensional rectangular area with a changing surface information on it, so it is a three dimensional object. Each surface information is discretely sampled from real world or calculated with different height field creation algorithms in order to mirror a real appearance of a landscape. Here the detail of terrain is decided with this sampling rate, where more detail requires more sampling. Densely sampled surfaces may be triangulated with millions of polygons where graphics hardware is unable to display such dense terrains in real time. In the future graphics cards will be able to process more and more polygons per second, but on the other hand the demand for using more complex and dense terrain models will also rise. This gap between the performance of graphics cards and the desire for displaying more complex models is not likely to disappear in the near future.

Terrain simulation applications should be evaluated with respect to some criteria. Application must achieve a given triangle count in a required time. 30 frames per second is a good time bound to maintain an optimized mesh containing thousands of triangles. Simplicity of algorithms is another criterion where rapid optimizing algorithms will clearly increase the frame rate and real time performance of the renderer. Visual quality is also important to have a visually satisfactory results at given triangle counts. As we mentioned above, approximation tolerance is a vital criteria and also a part of correct optimization of the algorithm. Another criterion is computer memory load where unnecessary memory usage may predict us to extend the terrain size or increase the terrain detail.

The general principle behind terrain rendering algorithms is that objects that are farther away require less geometry to be accurately represented than objects that are near. Similarly, objects that are very rough require more geometry to accurately represent than objects that are smooth. In an interactive terrain renderer, viewpoint can change over time and the amount of detail necessary to accurately represent any portion of the terrain can change over time as well. Therefore the representation of the terrain must be dynamic, allowing portions of the terrain that are currently far away to be rendered with few triangles but allowing the same portions of terrain to be rendered with more triangles if viewpoint moves closer. This kind of terrain rendering approach is called level-of-detail (LOD) rendering. Dynamically changing terrains have artifacts which we call pops. Applications also should reduce these disturbing temporal artifacts.

2 TERRAIN DATA REPRESENTATION

2.1 Geometrical Surface Description

At this section we are going to deal with the representation of the three dimensional terrain data geometrically. Terrain is a three dimensional surface so it can be represented with a surface function. We define such a surface explicitly with a bivariate vector function $p : R^2 \rightarrow R^3$ like $p(u,v) = [x(u,v), y(u,v), z(u,v)]$ where the two parameters *u* and *v* drive each component of the *p* vector [11]. Here clearly we create a mapping from two dimensional parameter domain to three dimensional mesh domain. Covering a sphere shaped object with an earth texture is a good example for parameterization.

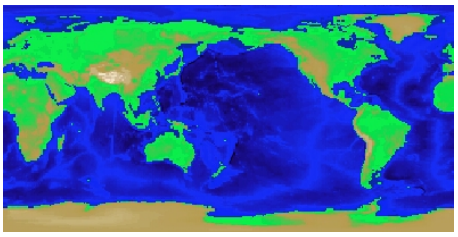


Fig. 2.1.1 Parameter domain.

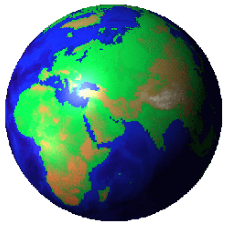


Fig. 2.1.2 Mesh domain.

Usually these functions are quantized at discrete intervals. Just like a discrete two dimensional image, we will have a set of points instead of a truly continuous surface. In order to preserve the surface's C^0 continuity in *R* we need to utilize interpolation methods. One of the simplest form of interpolation is the linear one, which is patching the hole between three neighboring points with a triangle. This kind of interpolation is also very convenient for hardware rendering.

2.2 Heightmaps (or Heightfields)

Despite the fact that the function described in the previous section gives us the most flexibility when defining a surface, in computer simulation terrain is not described such a generic surface. It is represented as a simple elevation data called heightmap.

By modifying the original surface representation above here we have a simpler definition of it, like $p(x,z) = [x, y(x,z), z]$, where $p : R^2 \rightarrow R^3$.

This function is much more constrained from the previous one, for example overhanging surfaces, like the one shown in Figure 2.1.1, cannot be modeled with it, because only one elevation value can be mapped to *y* values for each *x* and *z* value.

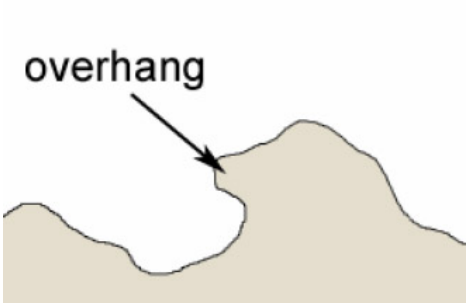


Fig. 2.2.1 Overhanging Surface.

So simply heightmap is the sampled representation of surface elevation values, the *y* values, at the vertices of a regularly spaced two dimensional grid in the vertical projection on the *xz*-plane (Figure 2.2.2).

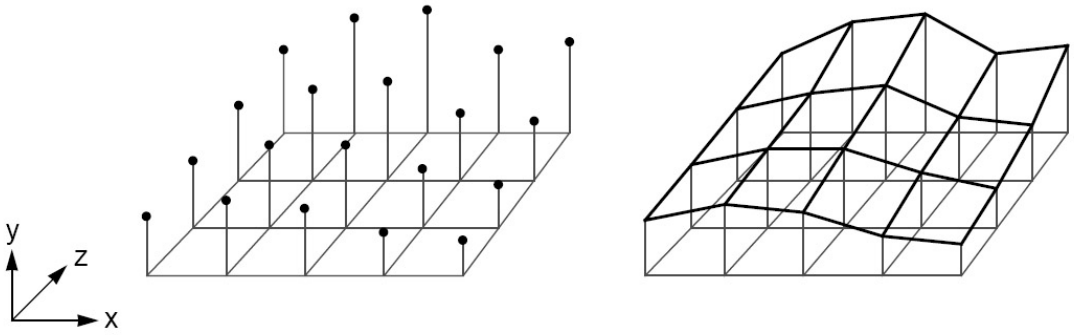


Fig. 2.2.2 Heightmap representation.

Unfortunately this simple approach is not practical when the heightfield data is large. The distance between adjacent grid points is called the resolution of the heightmap. Denser heightmap means a more detailed surface representation and larger heightfield data. For large heightfields the amount of geometry that must be rendered

exceeds the restrictions of the fastest 3D hardware currently available and restricts the real time performance of the terrain renderer.

2.3 Memory Restrictions

In many of the terrain simulation applications terrain covers an important amount of area. The detail of this area may change with the sampling ratio. Impressive visual success of terrain comes with densely sampled elevation values. Small changes in terrain may only be obtained when it is sampled in high ratios, but this kind of sampling requires more storage and memory capacity. For example, consider we have a heightfield which covers a 100 km $\times$ 100 km area on earth, if we sample it with 10m x 10m resolution we would have $10000 \times 10000 = 10^8$ data points. This is equal to a 95.36 MB of memory. If we sample it with 1m $\times$ 1m resolution we would have $100000 \times 100000 = 10^{10}$ data points, and this requires a 9.3 GB of memory.

It is waste of time try to map such a dense heightmap onto memory. One possible solution is dividing this complete heightmap into smaller blocks and load them one by one when they enter the view frustum of the user. By this way we can represent this entire terrain with desired detail and without having memory restriction problems.

Heightmap only gives us the three dimensional shape information of the surface. Behind this information a terrain must be covered with a texture. This texture will also bring another memory load for terrain. Simply, texture is a two dimensional image which is stretched along the terrain surface. Texture images may have various resolution levels and higher resolution is equal to more memory usage. So another paging system may be designed for terrain texturing and prevent memory overloads.

Up to date applications of terrain visualization need detailed heightmap and texture information. So not only the triangulation algorithm but also the management of the memory is very important in such applications. In following parts of this thesis I will introduce an effective paging method for both heightmap and texture management.

3 DISPLAYING TERRAIN

3.1 Overview

In order to display terrain we have to send the vertex information and appropriate connections to graphics hardware and let it render the terrain. At this point we are just talking about rendering the mesh neither containing the texture information nor being illuminated with a light source. There are two important points here, first one is which vertex information are we going to send, and the second one is how are these vertices be connected with each other. The answer of these two questions depends on the terrain triangulation algorithm.

One of the simplest solutions displaying a terrain is using Brute-Force algorithm [17]. This algorithm also provides the highest amount of detail possible. A 5 × 5 patch of Brute-Force terrain is given in Figure 3.1.1.

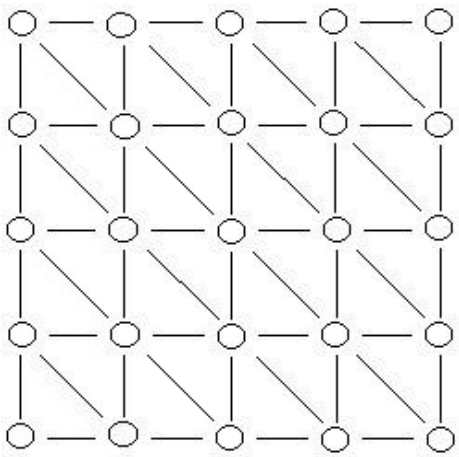


Fig. 3.1.1 A 5 x 5 patch of Brute-Force terrain.

In Figure 3.1.1 white circles represent the terrain vertices and the lines represent the connections between them. Each quadrilateral is built up out of 2 triangles. Unfortunately, Brute-Force algorithm is the slowest method that we can use in rendering. Even if we use the most proper rendering structure like triangle strips for each row of vertices in the patch we will not be able to obtain a real time implementation due to graphics hardware inabilities.

Generally terrain heightmap structure is in $(2^n+1)\times(2^n+1)$ form, so this will lead to $2^n \times 2^n$ quadrilaterals, and we will have to render $2\times 2^n \times 2^n$ triangles. Such an example is given below in Figure 3.1.2.

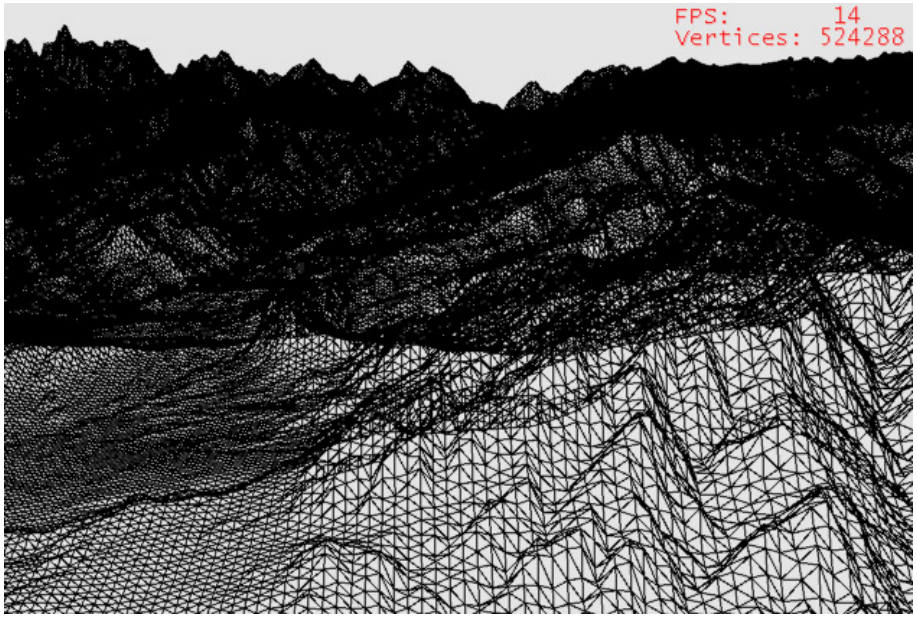


Fig. 3.1.2 A 513 x 513 patch of Brute-Force terrain.

For real time interactive simulation we need at least 15 frames per second (FPS), for continuous animation we need 25 FPS. Many of terrain simulation systems need to achieve 30 FPS minimally, but actually human brain is plausible to 60 FPS and more. Above in Figure 3.1.2 we rendered $2\times 512\times 512 = 524.288$ triangles and obtained a frame rate of 14 FPS. This is not satisfactory result for human brain even if it is a small patch of terrain in actual applications.

3.2 Level of Detail Concept

Geometric datasets, like terrains, can be too complex to render at interactive rates. That was what we see in previous section, where Brute-Force algorithm was not convenient for such applications. So we have to decrease the detail level of the mesh in such a manner that human visual system is not going to sense it much. In Figure 3.2.1 detail level of a model is reduced from left to right.

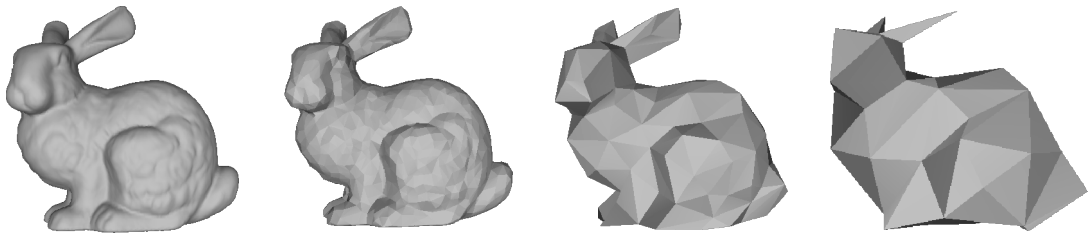


Fig. 3.2.1 Detail level of a model.

Human eye is less sensitive to distant objects. We can use this advantage and reduce the detail as object goes farther away. Figure 3.2.2 shows such an example.

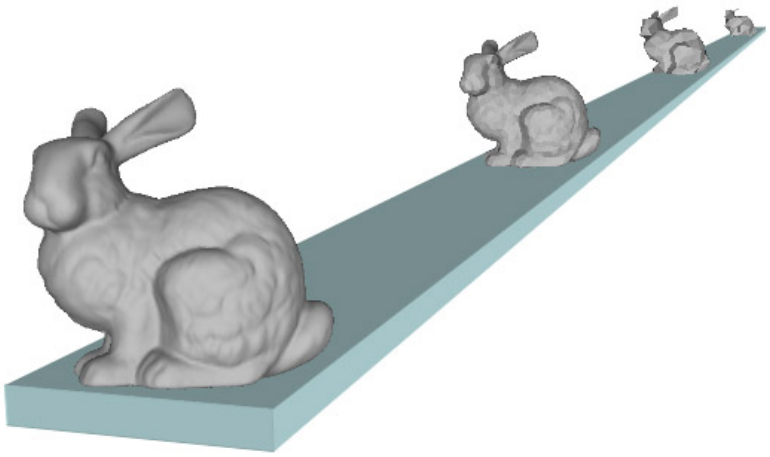


Fig. 3.2.2 Detail is reduced as object goes farther away.

Mesh reduction process is not an easy process and should be done according to some rules, because we do not want to loose rough details in the object even if it goes far away. For example in Figure 3.2.2, the ears of the rabbit are conserved and do not disappear as the detail decreases. We can use an error metric to decide if we need to drop or add some more detail to the mesh [19]. This error metric depends on the application but generally calculating the difference vector between two consecutive level-of-details in object space is a good approximation, where we call this difference vector object space relative error (Figure 3.2.3).

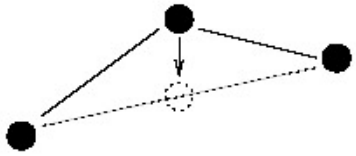


Fig. 3.2.3 Error introduced with difference vector.

This error metric does not take into account the perspective projection, i.e. what we see from computer screen. If we map this error from object space to screen space we obtain screen space relative error, which is more expensive to compute but visually more satisfactory.

Error term calculation process affects the algorithm performance as well. If we try to calculate this error term on a per vertex basis we will obviously lose performance, so sometimes it is better to calculate it for per triangle or for per different kind of mesh structure.

Basically there are two kinds of level-of-detail methods:

- Discrete Level of Detail (DLOD)
- Continuous Level of Detail (CLOD)

In discrete level-of-detail a mesh is statically partitioned into resolution levels. This different resolution levels are rendered according to the objects distance to the viewer. Resolution level creation is being done separately as a preprocess calculation. Partitioning the mesh in real time to these levels will decrease the performance, so geometric levels are constant before application runs. Main advantage of discrete level-of-details is that these statically partitioned meshes can be calculated easily for mesh optimization. Also they can be loaded to the local memory of graphical processing unit (GPU) for faster access and rendering. This method is generally used for distant objects and results are quite satisfactory, but in close view dependent applications it does not work. Because this method is a distance based method and in order to gain success in close up applications resolution level should

be increased. Even we increase the level count resolution changes will be apparent and disturbing popping effects will occur.

In terrain rendering discrete level-of-detail method will become a bottleneck when we fly closely on the terrain, so we must update the mesh view dependently. We can achieve this only in real time processing of the mesh. That is what continuous level-of-detail method does. It assembles the approximating mesh at runtime and gives a much better control over mesh approximation. Also resolution levels are not stored separately in the memory or any storage device. At every frame of the simulation mesh is revisited and updated according to the view distance and direction and sent to the rendering pipeline. Today, most of the terrain rendering algorithms use continuous level-of-detail method for its flexibility and advantage against discrete level-of-detail.

In continuous level-of-detail, mesh refinement is the focus point of the algorithm. While we use view-dependent refinement significant vertices should be selected according to the camera position and be rendered. We can define this process as a kind of mesh fitting problem on to a final mesh using some optimization methods to obtain a finer visual quality. Refinement is a kind of optimization. There two categories of refinement, bottom-up and top-down refinement. Bottom-up refinement starts from the most detailed mesh and decimates it to the appropriate mesh. It is also called decimation. Top-down refinement is just the opposite of bottom-up refinement. It starts from the coarser mesh and adds detail where necessary. Bottom-up refinement is size dependent, i.e. its calculation complexity is proportional to the mesh size, but it creates more optimal triangles. On the other hand complexity of top-down refinement does not greatly depend on the mesh size, but it creates less optimal triangulation than bottom-up refinement. For large datasets top-down refinement is more convenient and today many of the applications use this refinement method for its speed.

3.3 Common Problems

If we render terrain using a continuous level-of-detail algorithm, we must also deal with cracks and T-vertices. When two terrain tiles of different resolution meet on a common edge, due to the level difference cracks and T-vertices will occur. Even both tiles are aligned on that common edge these visual artifacts are unavoidable. In Figure 3.3.1 there are two tiles with different resolution.

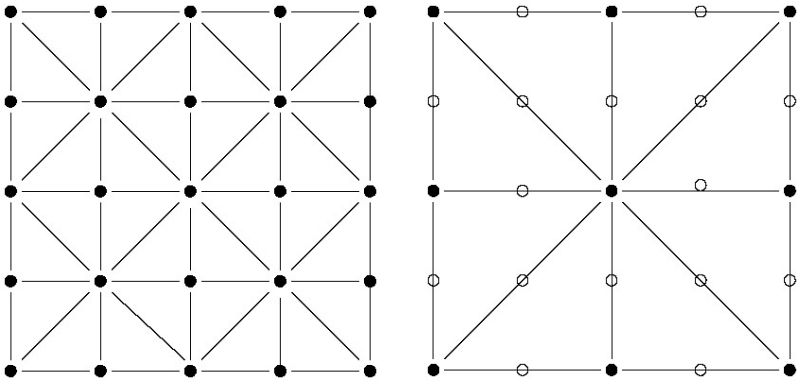


Fig. 3.3.1 Left tile level 0, right tile level 1.

Cracks and T-vertices will occur if these two tiles are joined together. In Figure 3.3.2, black squares indicate the T-vertices after joining. Because the left patch renders the exact height points at these T points, but the right patch is just getting the average of the height above it and the height below it. So, cracks will appear at these points.

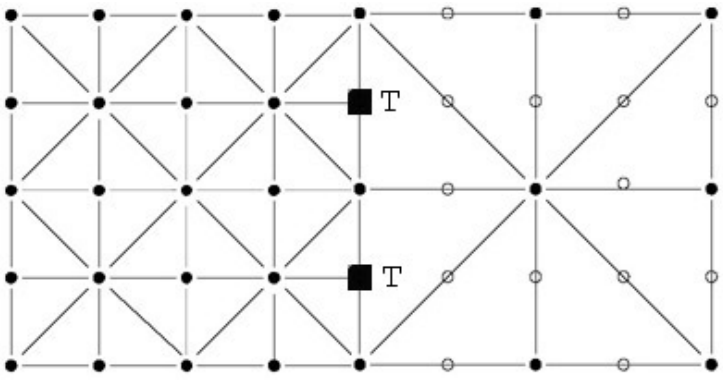


Fig. 3.3.2 Black squares indicate T-vertices.

In order to avoid these cracks we have to modify the geometry of one of these patches as they are rendered next to each other with different detail levels. We can

modify the patch with high level-of-detail and can avoid T-vertices and cracks. This modification is shown below in Figure 3.3.3.

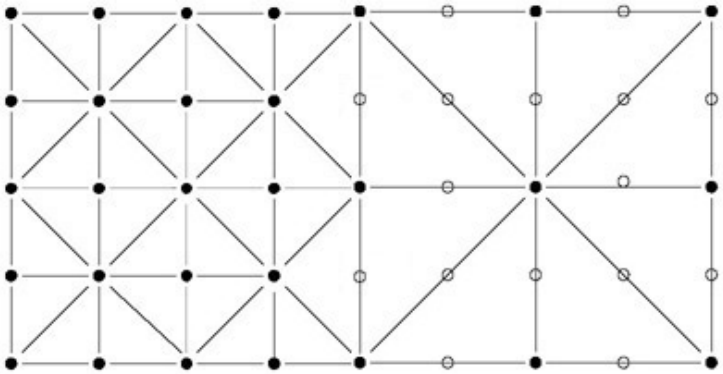


Fig. 3.3.3 Joined mesh without any T-vertices and cracks.

Another common problem in level-of-detail algorithms is popping. Popping occurs in transition between different detail levels due to screen space errors. In Figure 3.3.4 such a level reduction is represented.

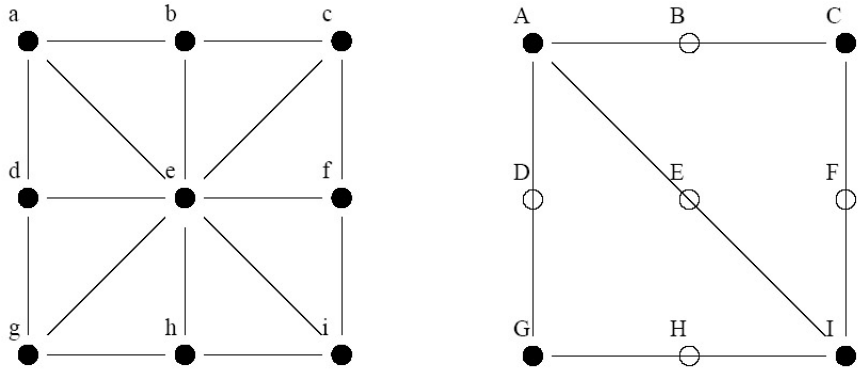


Fig. 3.3.4 High resolution mesh (left), after resolution reduction (right).

If we choose a low screen space error, the switch between two successive detail levels may be negligible, but this will conserve the triangle count in mesh patches and enough mesh reduction will not be obtained for rendering. Higher screen space error will let us decrease the triangle count, but on the other hand any switch between two successive detail levels will obviously create a popping effect.

Morphing means a smooth change between levels and may be a nice solution for this problem. For a sample patch in Figure 3.4.4 morphing calculations are given below [12]:

$$A = a \qquad D = v\frac{(a+g)}{2} + (1-v)d \qquad G = g$$

$$B = v\frac{(a+c)}{2} + (1-v)b \qquad E = v\frac{(a+i)}{2} + (1-v)e \qquad H = v\frac{(g+i)}{2} + (1-v)h$$

$$C = c \qquad F = v\frac{(c+i)}{2} + (1-v)f \qquad I = i$$

When morphing from a higher resolution to a lower resolution the values *b*, *d*, *e*, *f*, *h* are linearly interpolated between their original position and the values *B*, *D*, *E*, *F*, *H* respectively. This is possible changing *v* values from 0 to 1. Morphing from a lower resolution to a higher resolution is similar but the procedure must be inverted, i.e. *v* values must change from 1 to 0.

4 SURVEY OF COMMON ALGORITHMS

4.1 Overview

Terrain rendering has become a popular research area as the demand on displaying landscape areas grew up. Different algorithms developed having their advantages and drawbacks on this area. Also other research subjects such as terrain texturing or terrain lighting extended their algorithms according to these new rendering algorithms. In this section I will plainly explain the following algorithms:

- ROAM
- SOAR
- Geometrical MipMapping
- Quadtree

These algorithms basically implement different error metrics and different terrain partitioning structures from each other.

4.2 ROAM

ROAM stands for Real-time Optimally Adapting Meshes and developed by Mark Duchaineau [4]. In ROAM camera movement is considered to be continuous to take the advantage of frame to frame coherence. In flight simulators there is no sudden change in viewpoint and only few triangles differ in each frame, so this algorithm is basically developed for virtual flights.

ROAM uses triangle bintree structure for incremental refinement and decimation. The root triangle in this structure is defined to be $T = (v_a, v_0, v_1)$ and is at the coarsest level of subdivision, $l = 0$. At the next finest level, $l = 1$, the children of root are defined by splitting the root along an edge formed from its apex vertex v_a to the midpoint v_c of its base edge (v_0, v_1). The left child of T is $T_0 = (v_c, v_a, v_0)$, while

the right child of T is $T_1 = (v_c, v_1, v_a)$. The rest of the triangle bintree is defined by recursively repeating the splitting process. This can be seen in Figure 4.2.1.

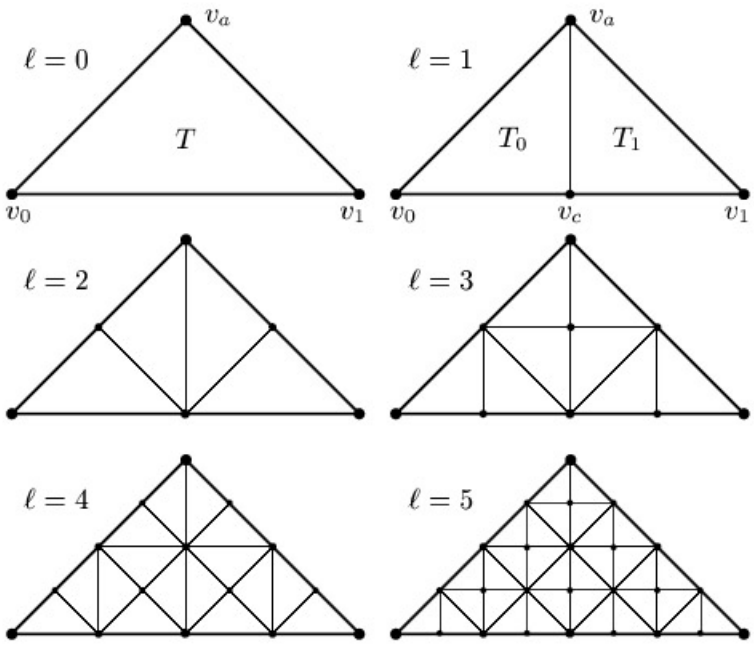


Fig. 4.2.1 Splitting process of bintree.

In bintree triangulation neighbors of a triangle is either from the same bintree level *l*, or from the next finer level $l+1$, or from the next coarser level $l-1$ for base neighbors. A triangle T in a triangulation cannot be split immediately when its base neighbor T_B is from a coarser level. To force T to be split T_B must be forced to split first, which may require further splits in a recursive sequence (Figure 4.2.2). This is called forced split, and it is necessary to avoid cracks and T-vertices.

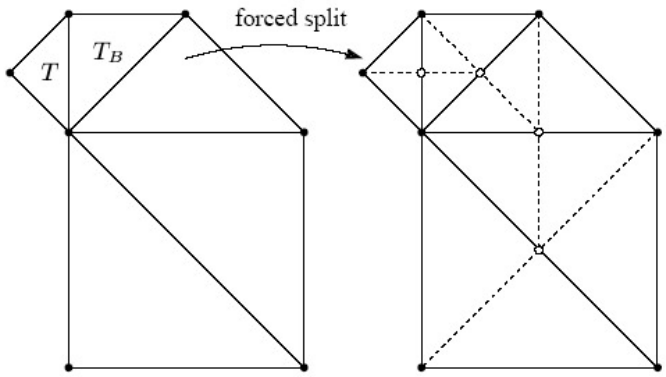


Fig. 4.2.2 Forced splitting to avoid cracks.

ROAM uses two priority queues that splits and merges the mesh. The queues are filled with triangles, sorted by their projected screen space error. At each frame some triangles are split and some are merged according to camera viewpoint. If priorities change smoothly and slowly, then the optimal triangulations for any two consecutive frames will tend to be similar to one another. So performance will be enhanced if we use triangulation T_{f-1} as a starting point to build triangulation T_f. It is also possible to stop the refinement at any time, since the mesh is progressively updated, and obtain stable frame rates.

Splitting and merging are expensive operations for real time applications. To do these from the beginning each frame is a bottleneck, so incremental refinement is considered here. But if you move your camera faster, incremental refinement will also not help to solve this problem. As a result ROAM is effective for only smooth changes in the viewer location.

4.3 SOAR

SOAR stands for Stateless One-pass Adaptive Refinement and developed by Peter Lindstrom and Valerio Pascucci [1]. SOAR's refinement algorithm generates a new mesh from scratch every frame, thus it avoids ROAM's biggest problem. But it is not possible to stop the refinement at any time. We have to finish refinement in order to obtain a continuous surface. This is not a big problem for SOAR, because it is possible to adjust the error threshold between frames and thus adaptively change refinement parameters and obtain a stable frame rate. SOAR uses longest edge

bisection of right isosceles triangles, i.e. each triangle is subdivided by its hypotenuse creating two smaller right isosceles triangles. This scheme is shown in Figure 4.3.1.

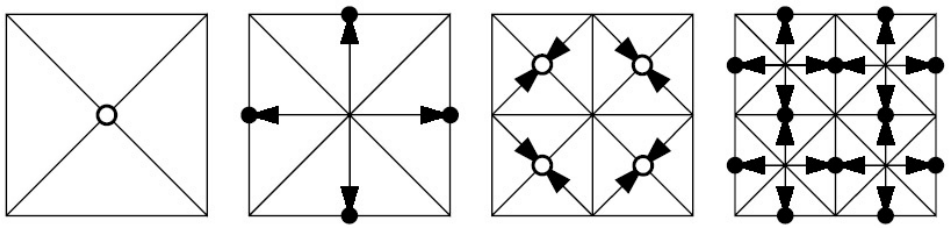


Fig. 4.3.1 Edge bisection hierarchy.

Starting with a coarse base mesh, which can consist of two or four triangles, recursive refinement is made. The refinement criterion is generally based on whether the triangle approximates the corresponding part of the original high resolution mesh well enough. The new vertices introduced by each subdivision lay on a rectilinear grid, and this makes it easy to map a surface on it. If n is the number of refinement levels, the dimensions of the underlying grid are constrained to $2^{n/2}+1$ vertices in each direction.

The mesh produced by edge bisection can be represented as a Directed Acyclic Graph (DAG) of its vertices. Thus, all inner vertices are connected to four children and have two parent vertices. Boundary vertices have two children and one parent as in Figure 4.3.1. At the update process of the mesh in order to avoid cracks and T-vertices not only the active vertices but also their parents and also ancestors should be activated recursively. Visiting each parent and ancestor of a considered vertex and activating them may solve this problem. But this is, as we seen before, is a very costly operation.

SOAR uses bounding spheres to create nested hierarchies. Each vertex is assigned a bounding sphere, and each bounding sphere contains all of its descendant vertices' bounding spheres (Figure 4.3.2).

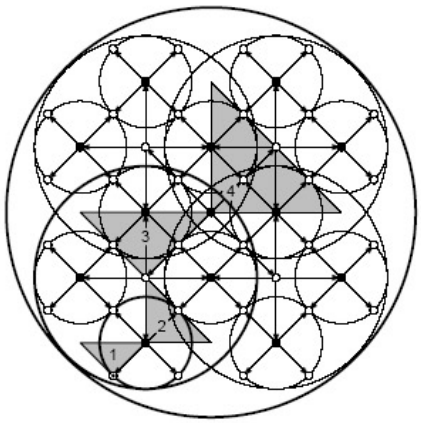


Fig. 4.3.2 2D analogue of the nested sphere hierarchy used for refinement and view culling.

Here, if we project every vertex's object space error from the point on its bounding sphere's surface that is closest to the camera, then the resulting projected error will be nested. It is obvious that this projected error might be higher than the original error. But using this hierarch, large blocks can be reduced without visiting the descendents. It also enables efficient view frustum culling, if a sphere is totally outside the frustum vertex is culled with all of its descendants. These computations are done as a pre-processing step and by this way real-time processing cost is decreased.

SOAR renders refined mesh in such a manner that it generates a continuous triangle strip, and passes this to hardware. Each triangle is processed from left to right as seen in Figure 4.3.3.

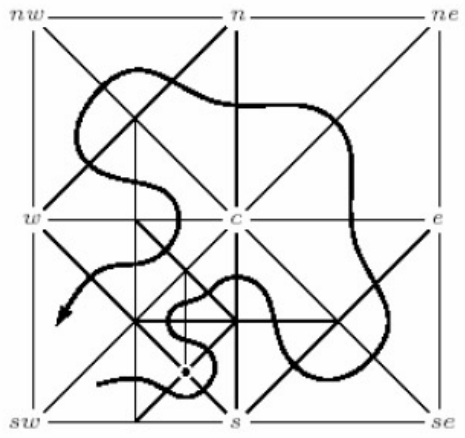


Fig. 4.3.3 Triangles are rendered from left to right.

4.4 Geometrical MipMapping

Generally, in continuous level-of-detail algorithms more detail is added where it is needed. Especially more chaotic terrain patches are densely triangulated than other smooth patches. But not all algorithms require more detail for such chaotic areas of terrain, and Geometrical Mipmapping (Geomipmapping) is such an algorithm that does not distribute more triangles to areas that require more detail.

Geomipmapping is developed by Williem H. de Boer [7], and it is similar to mipmapping technique for textures [10]. In texture mipmapping, a chain of textures

are created, where first item is the actual texture and succeeding item being the half resolution downsampled of the previous item. Number of items in the chain is level depth. When a texture is at certain distance from the viewpoint, the proper level in the chain is chosen and displayed instead of the high resolution version. Boer extended this concept in three dimensions and calculated mipmaps by scaling down the terrain block. For example, in Figure 4.4.1, level 0 is the most detailed patch, as this patch comes farther away from the viewer location, it changes to level 1, and even the farther away it reaches to level 2, which is the lowest level of detail.

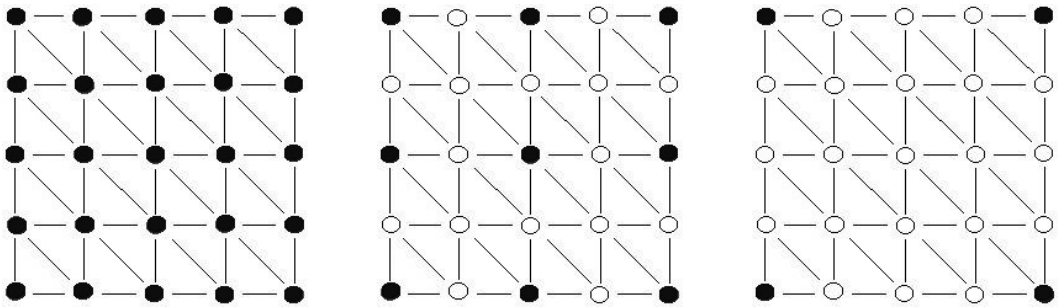


Fig. 4.4.1 Patch levels from high to low.

Geomipmapping is composed of a series of patches, i.e. complete terrain is cut up into terrain blocks, which have a fixed size of the form $\left(2^n+1\right)\times\left(2^n+1\right)$. Boer used 17×17 size of terrain blocks for a 257×257 sized terrain grid. It is also very simple to prevent cracks in geomipmapping , basically we omit to render the vertices from the more detailed patch, as in Figure 3.3.3.

Two different methods are suggested to choose the appropriate geomipmap levels. One is using a fixed, preset distance (*d*) value for each of the geomipmap levels. This method works well if d is large enough and popping effects does not disturb anyone, but in close views of terrain sudden level changes will cause sensitive popping artifacts. Another way is to choose these level changes according to screen space error. The projection of object space error, in Figure 3.2.3, is called screen space error, which is eventually what the user will notice. If we call this error ε, when ε exceeds a certain threshold τ the error will be too noticeable and therefore it is not permitted to switch to a higher level until ε is smaller than τ. Every geomipmap level consist of several of these errors, taking the resulting ε from $\max\{\varepsilon_0,\Lambda\ ,\varepsilon_{n-1}\}$,

we can find the worst-case scenario. If this error value is lower than τ, then all the geomipmap's error values will be smaller than τ. If this error value is higher than τ, then there will be at least one error that is too noticeable.

Even if we use screen space error to select geomipmap levels and decrease the popping effect, there will still be some occasional popping effects in the mesh, especially when τ is set to a high value. Finding a proper τ value for each particular application is another problem, because this value is unique for each application. Boer suggested a method called trilinear geomipmapping, which is the same technique as geomorphing to decrease popping effects visually.

In Figure 4.4.2 you can see a rendered scene using geomipmapping.

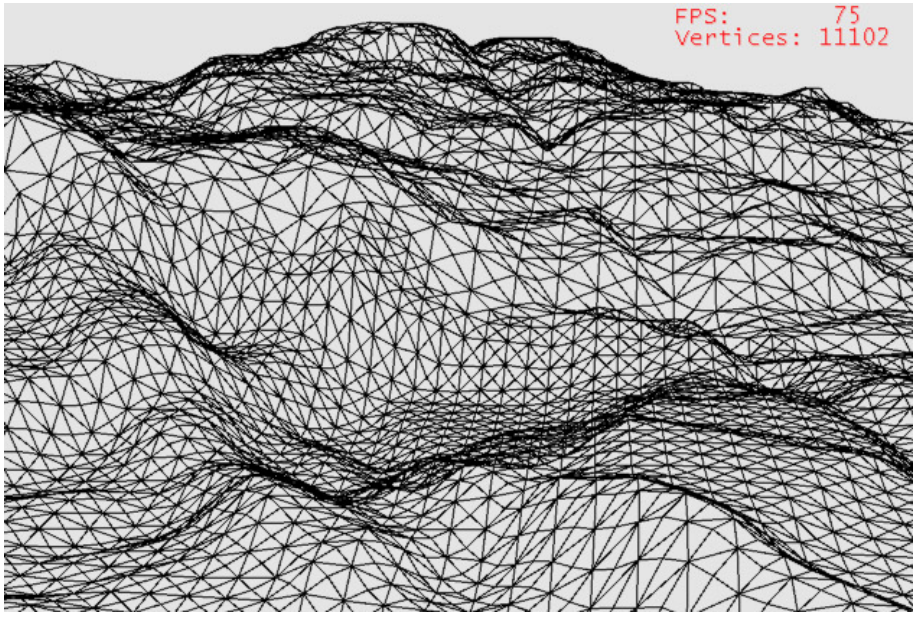


Fig. 4.4.2 A 513 x 513 patch of GeoMipMapped terrain.

4.5 Quadtree

Quadtree structure is commonly used in image processing and computer vision, especially in binary region representation or region splitting algorithms. A quadtree is a two dimensional structure and it is very convenient for terrain rendering algorithms. One of the most popular implementations of quadtree algorithm for terrain is Stefan Röttger's implementation [6]. His algorithm uses a top-down

strategy to create a triangulation. Vertex removal is performed depending on its distance to the point of view as well as local surface roughness. This structure also allows very efficient clipping.

The considered heightfield structure is of size $(2^n+1)\times(2^n+1)$. So a simple triangulation using quadtree is shown in Figure 4.5.1.

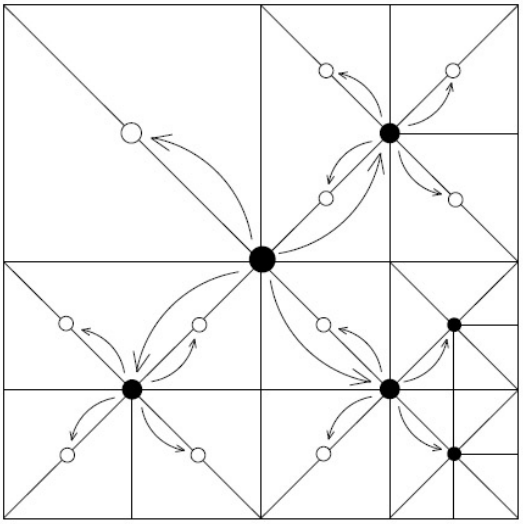


Fig. 4.5.1 A sample triangulation of a 9x9 heightfield.

The quadtree in Figure 4.5.1 can be represented by a boolean matrix as in Figure 4.5.2.

$$\begin{pmatrix} ? & ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & 0 & ? & 0 & ? \\ ? & ? & 0 & ? & ? & ? & 1 & ? & ? \\ ? & ? & ? & ? & ? & 0 & ? & 0 & ? \\ ? & ? & ? & ? & 1 & ? & ? & ? & ? \\ ? & 0 & ? & 0 & ? & 0 & ? & 1 & ? \\ ? & ? & 1 & ? & ? & ? & 1 & ? & ? \\ ? & 0 & ? & 0 & ? & 0 & ? & 1 & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? \end{pmatrix}$$

Fig. 4.5.2 Boolean matrix representing the quadtree.

This matrix contains a boolean value for every possible node center in the quadtree. Here, each 0 and 1 represents if a node center is enabled or not. Question marks are considered to be the nodes that will not be visited during rendering process and stop

the subdivision of the mesh. This is a top-down algorithm so rendering quality does not depend on heightfield size.

Hegihtfield is rendered using triangle fans, which makes this process to solve simply. The quadtree is traversed recursively and the corresponding matrix entries are set. After reaching a leaf the triangle fan is rendered. Rendering a triangle fan is done fully or partially. This is necessary to prevent cracks, and triangle fans are very suitable for this process.

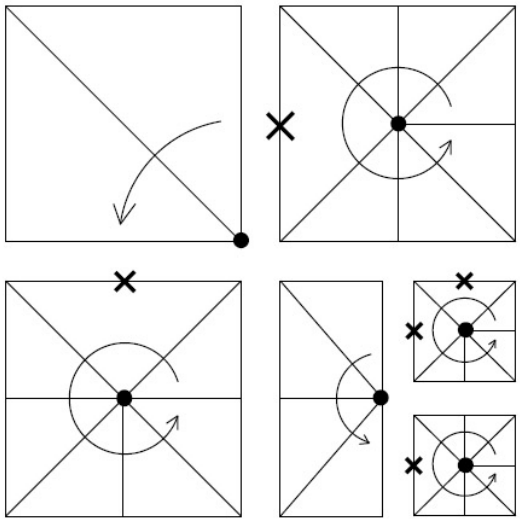


Fig. 4.5.3 Recursively generated triangle fans.

During the rendering process if the neighboring node is not subdivided to the same level, the center vertex on the shared edge is skipped. In Figure 4.5.3 we can see the rendering process of quadtree in Figure 4.5.1. Here, crosses indicate skipped vertices for crack protection. The recursive rendering function is called only seven times each for one triangle fan.

The triangulation is build by recursively descending the quadtree and at each node a boolean subdivision criterion is evaluated to subdivide the mesh or not. Röttger used two different aspects for criterion. First one is to decrease the resolution of the mesh as the distance from the viewer increases, and the second one is to increase the resolution for regions of high surface roughness. The second criterion is a costly algorithm, in which the complete mesh should be analyzed and roughness values should be calculated. So this step is done as a pre-processing step of the algorithm.

In Figure 4.5.4 you can see a rendered scene using quadtrees.

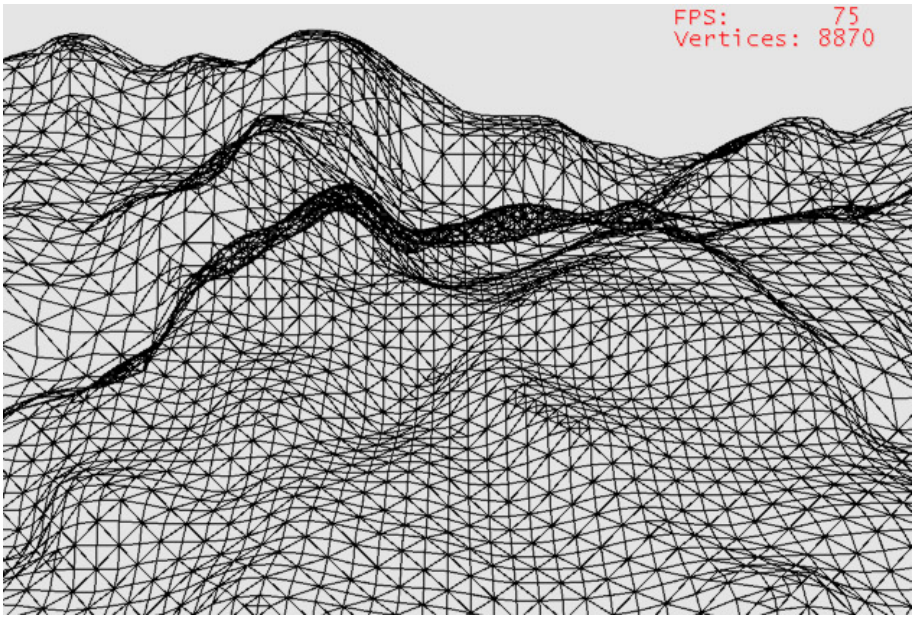


Fig. 4.5.4 A 513 x 513 patch of quadtree terrain.

5 A PROPER MESHING ALGORITHM FOR PAGING

5.1 Overview

In the previous section we covered some very famous rendering algorithms and their properties, and in this section we are going to make a decision what kind of a rendering algorithm is convenient for large scaled terrains. Large scale may have two meanings: a terrain which is very densely sampled and visualized, and a terrain which covers a huge geographical area. In our implementation we deal with the second one, i.e. huge geographical area.

Developed algorithms can work with quite large hightmap datas, but this is also not enough for what we call huge. While we render a heightfield using one of the algorithms above we will have to take the complete map into memory. But in such a situation we will suffer from memory restrictions of computer hardware and this will not work for huge sets, so we will have to do paging. To have a good paging performance, our rendering algorithm should load and create the mesh hierarch as fast as it can. So a pre-processing step for each heightmap block will reduce this performance.

There is also important to balance the allocation structure size for mesh triangulation and creation speed. Allocating nodes of meshes one by one according to their importance is a good method for memory organization, but this allocation will reduce the performance of the algorithm. Allocating the complete structure from the beginning may increase this allocation speed, but by his way we cannot predict the enough node size for triangulation and we create them all from beginning.

We have to acquire strict frame rates at least 30 FPS for a medium quality of simulation. For a good quality we have to obtain 60 FPS. Triangle count for rendering is the main reason which decreases this rate, so we have to balance it well not decreasing the visual quality of the terrain. Horizontal view angles contain more triangles than vertical ones, so we have different frame rates when we look in different directions even if we are at a fixed point of view.

Considering all of these artifacts we prepared a rendering algorithm for a heightmap block, which is efficient for loading, deleting, updating and rendering for each tile of a paged system for a very large scaled terrain.

5.2 Updating and Rendering Algorithm

Taking into account the reasons in the overview section, the best rendering algorithm is a quadtree algorithm for very large scaled systems. Allocation of each node dynamically will reduce the loading time of terrain, so using static arrays as in [6] is an efficient method. On run time just modifying elements of this array, we enable or disable vertices. Because of the structure of the quadtree, our heightmap also should be in a size of $\left(2^n+1\right)\times\left(2^n+1\right)$. This may be considered as a restriction but on the other hand using such a structure improves the management of the paging system.

Generally a rendering algorithm consists of two parts. First one is an updating process and the second one is rendering process. Building the triangulation by recursively descending the quadtree is done at updating process. The basic concept in updating the terrain is to decrease the resolution as the distance from the viewer increases. Following inequality guarantees this condition:

$$\frac{l}{d} < C \qquad (5.1)$$

In equation (5.1), l represents the distance from the center of the current node to the camera eye, d is the edge length of the current node, and C is a parameter which controls the overall detail for the mesh, i.e. is a configurable quality parameter or minimum global resolution. As C increases, the total number of vertices per frame grows quadratically (Figure 5.2.1).

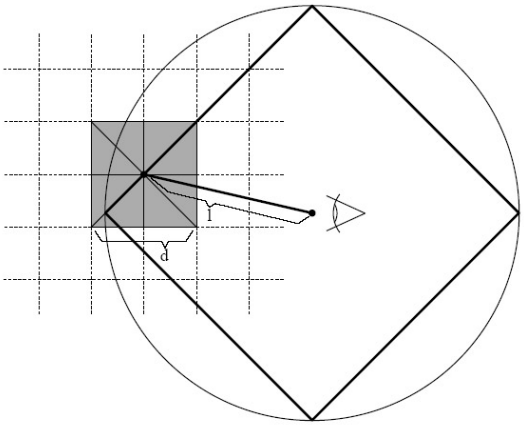


Fig. 5.2.1 Distance versus size of quadtree cells.

Distance measurement can be done in two ways. First one is $L^2 - norm$ which is:

$$l = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{5.2}$$

$L^2 - norm$ is a quadratic form of equation, and its solution contains a square root operation, which becomes costly if we calculate it for each considered node. So we used a second measurement criterion which is called $L^1 - norm$:

$$l = |x_2 - x_1| + |y_2 - y_1| + |z_2 - z_1| \tag{5.3}$$

$L^1 - norm$ is rather linear than $L^2 - norm$. Using $L^1 - norm$ we do not calculate square of each individual component and get rid of a square root calculation. If we represent the node subdivision criteria in a different way we get the following Equation:

$$f = \frac{l}{d \times C \times MAX(c,1)} \tag{5.4}$$

The constant C determines the minimum global resolution, whereas the constant c is desired global resolution. By adjusting this desired global resolution constant, we can arrange the system load and get a constant frame rate. In Equation (5.4), f is our final

step in deciding whether we want to subdivide a node. After calculating the value of *f*, we can subdivide it after testing a simple condition:

```
if (f < 1)
    subdivide_node();
```

In [6], Röttger suggested a second criterion to increase the resolution for regions of high surface roughness. This requires an extra work and CPU usage and is not essential. We can use it only if we want more detail to rough regions. This criterion simply tries to minimize the vertex error due to hierarchy dropping. One level of hierarchy reduction in a node extracts a new error which is introduced at the center of the node and the four midpoints of its edges. This can be seen in Figure 5.2.2.

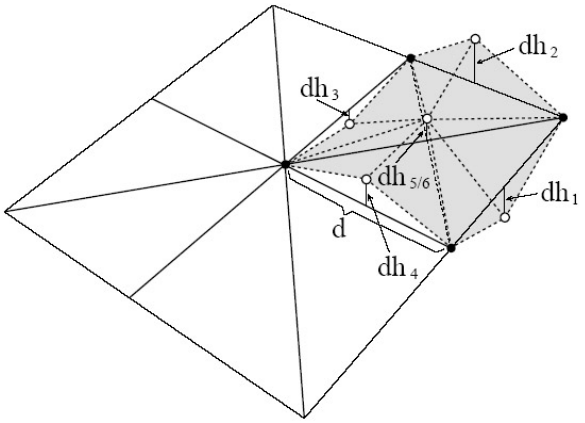


Fig. 5.2.2 Points introducing error.

For every node a value called $d2$ is calculated taking the maximum of the absolute values of the elevation differences dh_i as a preprocessing step.

$$d2 = \frac{1}{d} \max_{i=1..6} |dh_i| \tag{5.5}$$

In order to make sure that surrounding nodes does not differ by more than one level of detail a bottom-up approach is followed. Starting from the highest level-of-detail mesh the local $d2$ values of each node is calculated and propagated up to the quadtree. If there is a difference in the level-of-detail that is greater than one, the $d2$

values of coarser nodes are adjusted using the previous $d2$ values. After calculating $d2$ values as a preprocessing step the Equation (5.4) is updated with these values as in Equation (5.6).

$$f = \frac{l}{d \times C \times MAX(c \times d2, 1)} \tag{5.6}$$

As can be seen in Equation (5.6) f values are only weighted with $d2$ values, so updating process step does not bring any processing delay more than a simple multiplication. But for interactive very large scaled terrain systems the paging process may be affected due to the preprocessing step for $d2$ values.

One major method to improve the performance of rendering and updating is to perform view frustum or simply frustum culling. The view frustum is the volume of space that includes everything that is currently visible from a given viewpoint. View frustum consists of six planes as in Figure 5.2.3.

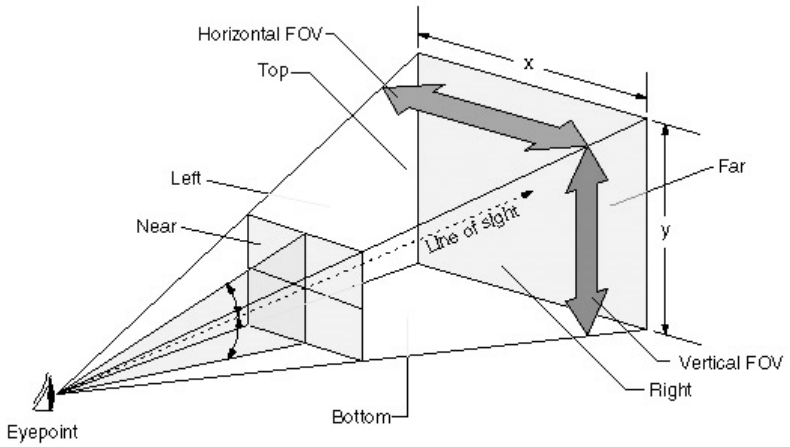


Fig. 5.2.3 View frustum pyramid.

Frustum culling is to determine whether an object is in this viewing pyramid or not, and as a result of this calculation to decide to send that object to the rendering pipeline. Taking into account a large scaled terrain such a culling will save both CPU and GPU processing time. Basically we test each path against the frustum. If we create an Axis-Aligned Bounding Box of that patch we can test it against the frustum

easily. If it is out of the frustum we skip other calculations over that patch and continue with another patch. Frustum culling is done every frame, at each cycle view frustum is calculated than frustum test is applied for each terrain patch. Even if it looks as an expensive process it will make the algorithm much faster.

The pseudocode for updating a terrain is given in Table 5.2.1.

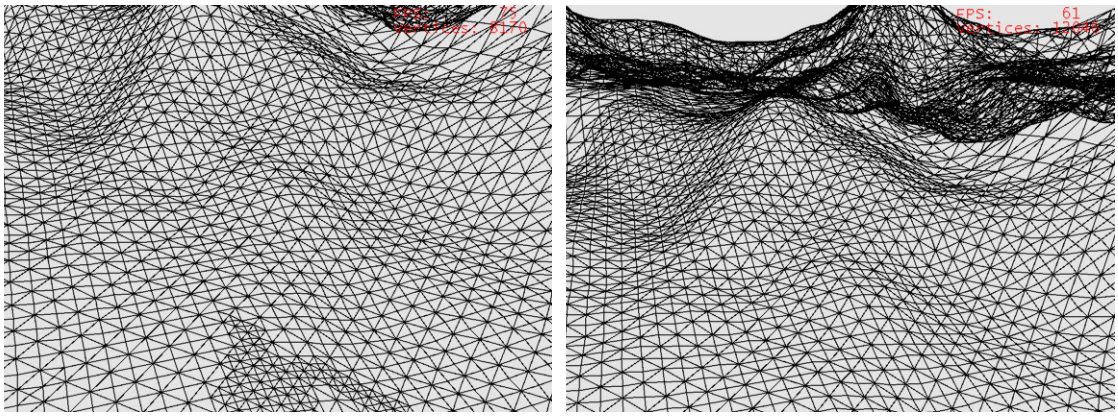
Table 5.2.1 Terrain refining algorithm pseudocode.

```
Function Update()
1  centerNode ← (edgeLength – 1) / 2;
2  Call RefineNode(centerNode, centerNode, edgeLength);

Function RefineNode(centerX, centerZ, edgeLength)
1  centerY ← Call GetYFromPatch(centerX, centerZ);
2  result ← Call CubeFrustumTest(centerX, centerY, centerZ, edgeLength);
3  if result == FALSE then
4      Call NoSubdivide(centerX, centerY);
5      return;
6  viewDistance ← Call L1Norm(cameraX, cameraY, cameraZ, centerX, centerY, centerZ);
7  f = viewDistance / (edgeLength * C * MAX(c, 1));
8  if f < 1 then
9      Call Subdivide(centerX, centerY);
10     if edgeLength <= 3 then
11         return;
12     else
13         childOffset ← edgeLength / 4;
14         childEdgeLength ← edgeLength / 2;
15         Call RefineNode(centerX – childOffset, centerZ – childOffset, childEdgeLength);
16         Call RefineNode(centerX + childOffset, centerZ – childOffset, childEdgeLength);
17         Call RefineNode(centerX – childOffset, centerZ + childOffset, childEdgeLength);
18         Call RefineNode(centerX + childOffset, centerZ + childOffset, childEdgeLength);
19         return;
20 else
21     return;

Function NoSubdivide(centerX, centerY)
1  quadMatrix[centerX][centerY] ← FALSE;

Function Subdivide(centerX, centerY)
1  quadMatrix[centerX][centerY] ← TRUE;
```

The algorithm in Table 5.2.1 has one basic function which is RefineNode. This function is a recursive function and it traverses from top to down of the quadtree. At each call one node is subdivided into for children nodes and RefineNode function is also called for them. Of course subdivision is done if subdivision criterion is

achieved. Algorithm stops after reaching the smallest and most detailed node which has an edge length of 3. By this way complete quadtree matrix is filled with appropriate values.

Frustum culling causes us to render only triangles which are in the viewing pyramid. But this triangle count may change according to the viewing direction of our eye point. Rendered triangle number may increase as look at vector becomes more parallel to x-z coordinate plane of the terrain. Especially if we use the roughness criterion as in Equation (5.6) the rough areas are going to be tessellated more than smooth ones, so horizontal look at these areas will case extremely high triangle counts than vertical looks. As a result, at constant position, changing the viewing direction may decrease or increase the frame rate. Vertical looks cause less triangles to be rendered than horizontal looks. Following two figures are taken from the same camera position, but with different orientation.



(a) Vertical view (b) Horizontal view

Fig. 5.2.4 Different views of same terrain.

As can be seen from Figures 5.2.4.a and 5.2.4.b horizontal view renders more triangles than vertical one. The frame rate of vertical look is 75 FPS and horizontal is 61 FPS, also rendered triangle number in horizontal view is greater than the vertical one. We can arrange our node weights according to the camera orientation and balance our frame rate. Let $v_L = [x_L \quad y_L \quad z_L]$ be our normalized look at vector and $v_B = [x_B \quad 0 \quad z_B]$ be our normalized base vector, which is on the two dimensional heightfield planes. The cosine of angle between these two vectors gives us the deviation between them as in Equation (5.7).

$$\cos\theta = v_L \bullet v_B \tag{5.7}$$

θ is the pitch value of our look at vector and may be any value between 0 and 360 degrees, so regarding cosine values change between -1 and 1. Taking the absolute value of this cosine value we restrict them to be between 0 and 1, then we can apply the following new metric in order to arrange weight values according to look angle.

$$C_N = \frac{C}{\left(1+\left|\cos\theta\right|\right)^4} \tag{5.8}$$

Increase in θ means a more vertical look at the terrain. While modifying the f constraint in Equation (5.4) with this metric we can reduce the triangle count at horizontal looks.

$$f = \frac{l}{d \times C_N \times MAX(c,1)} = \frac{\left(1+\left|\cos\theta\right|\right)^4 \times l}{d \times C \times MAX(c,1)} \tag{5.9}$$

Updating process only marks the quadtree matrix with proper values. In order to see the terrain mesh on screen we have to send it to rendering pipeline with proper mesh connections. After that showing it on screen belongs to GPU. Rendering a quadtree is also a recursive and long process. The pseudocode of rendering algorithm is given below in Table 5.2.2.

Table 5.2.2 Terrain rendering algorithm pseudocode.

```
Function Render()
1  centerNode ← (edgeLength – 1) / 2;
2  Call RenderNode(centerNode, centerNode, edgeLength);

Function RenderNode(x, z, edgeLength)
1  edgeOffset ← (edgeLength - 1) / 2;
2  adjOffset ← edgeLength – 1;
3  renderFlag ← quadMatrix[x][z];
4  if renderFlag == 0 then
5      return;
6  if edgeLength <= 3 then
7      glBegin(GL_TRIANGLE_FAN);
```

```
 8          glVertex(x, z);
 9          glVertex(x – edgeOffset, z - edgeOffset);
10          if quadMatrix[x][z – adjOffset] != 0 then
11              glVertex(x, z – edgeOffset);
12          glVertex(x + edgeOffset, z – edgeOffset);
13          if quadMatrix[x + adjOffset][z] != 0 then
14              glVertex(x + edgeOffset, z);
15          glVertex(x + edgeOffset, z + edgeOffset);
16          if quadMatrix[x][z + adjOffset] != 0 then
17              glVertex(x, z + edgeOffset);
18          glVertex(x - edgeOffset, z + edgeOffset);
19          if quadMatrix[x – adjOffset][z] != 0 then
20              glVertex(x - edgeOffset, z);
21          glVertex(x - edgeOffset, z - edgeOffset);
22      glEnd();
23  else
24      childOffset ← (edgeLength - 1) / 4;
25      childEdgeLength ← (edgeLength + 1) / 2;
26      fanCode ← quadMatrix[x + childOffset][z + childOffset] * 8;
27      fanCode ← fanCode | quadMatrix[x - childOffset][z + childOffset] * 4;
28      fanCode ← fanCode | quadMatrix[x - childOffset][z - childOffset] * 2;
29      fanCode ← fanCode | quadMatrix[x + childOffset][z - childOffset];
30      if fanCode == 15 then
31          Call RenderNode(x – childOffset, z – childOffset, childEdgeLength);
32          Call RenderNode(x + childOffset, z – childOffset, childEdgeLength);
33          Call RenderNode(x – childOffset, z + childOffset, childEdgeLength);
34          Call RenderNode(x + childOffset, z + childOffset, childEdgeLength);
35          return;
36      if fanCode == 5 then
37          glBegin(GL_TRIANGLE_FAN);
38              glVertex(x, z);
39              glVertex(x + edgeOffset, z);
40              glVertex(x + edgeOffset, z + edgeOffset);
41              glVertex(x, z + edgeOffset);
42          glEnd();
43          glBegin(GL_TRIANGLE_FAN);
44              glVertex(x, z);
45              glVertex(x - edgeOffset, z);
46              glVertex(x - edgeOffset, z - edgeOffset);
47              glVertex(x, z - edgeOffset);
48          glEnd();
49          Call RenderNode(x – childOffset, z + childOffset, childEdgeLength);
50          Call RenderNode(x + childOffset, z – childOffset, childEdgeLength);
51      if fanCode == 10 then
52          glBegin(GL_TRIANGLE_FAN);
53              glVertex(x, z);
54              glVertex(x, z + edgeOffset);
55              glVertex(x - edgeOffset, z + edgeOffset);
56              glVertex(x - edgeOffset, z);
57          glEnd();
58          glBegin(GL_TRIANGLE_FAN);
```

```
59              glVertex(x, z);
60              glVertex(x, z - edgeOffset);
61              glVertex(x + edgeOffset, z - edgeOffset);
62              glVertex(x + edgeOffset, z);
63          glEnd();
64          Call RenderNode(x + childOffset, z + childOffset, childEdgeLength);
65          Call RenderNode(x - childOffset, z – childOffset, childEdgeLength);
66      if fanCode == 0 then
67          glBegin(GL_TRIANGLE_FAN);
68              glVertex(x, z);
69              glVertex(x – edgeOffset, z - edgeOffset);
70              if quadMatrix[x][z – adjOffset] != 0 then
71                  glVertex(x, z – edgeOffset);
72              glVertex(x + edgeOffset, z – edgeOffset);
73              if quadMatrix[x + adjOffset][z] != 0 then
74                  glVertex(x + edgeOffset, z);
75              glVertex(x + edgeOffset, z + edgeOffset);
76              if quadMatrix[x][z + adjOffset] != 0 then
77                  glVertex(x, z + edgeOffset);
78              glVertex(x - edgeOffset, z + edgeOffset);
79              if quadMatrix[x – adjOffset][z] != 0 then
80                  glVertex(x - edgeOffset, z);
81              glVertex(x - edgeOffset, z - edgeOffset);
82          glEnd();
83          return;
```

The function is simple and starts to check whether the current node is of the highest level of detail or not. If so we just render it and return, we also check the detail level of surrounding nodes, if they are a lower level of detail we skip the vertex on that side. If the current node is not of highest level of detail we calculate what kind of triangle fans to draw for the current node according to its neighboring vertices. From the method we can see that rendering is also a top-down traversing method.

During rendering we used triangle fans for sending the geometry to pipeline. But this method is not the most effective one. Lindstrom in [1] and [2] used triangle strips for rendering. He arranges vertices in such a manner that a continuous triangle strip is created and sent to the pipeline. This is also the best method for rendering but not appropriate for detailed texturing. We will cover this problem later in detail at texturing section.

6 THE PAGING ALGORITHM

6.1 Overview

In the previous section we described a proper algorithm for paging. Many of the algorithms in the literature have been developed under the concept of a single paged system, and no matter what happens their performance will be vitally affected if we apply a paging system to them. Even if we arrange the updating and rendering algorithm for paging, simple management of pages will reduce the performance. A paging system is necessary when we fly over a very large scaled terrain, which cannot be mapped into the memory all in one pass. Terrain blocks have to be loaded and freed from an external storage device according to the camera position and must also obtain the desired visual quality for the viewer. Such a paging involving data access on secondary storage is often called a dynamic scene management mechanism.

A common paging method suggested by R. Pajarola in [13] is called windowing. Whenever the coordinates of the visible scene window W change to represent a new view W' onto the world, the outdated data of region W/W' has to be discarded and the newly visible region W'/W must be loaded from the external storage. This management method is illustrated in Figure 6.1.1.

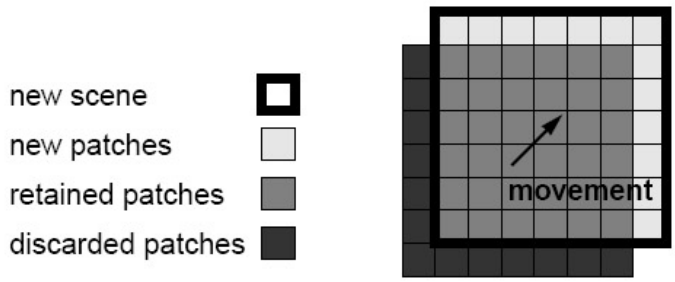


Fig. 6.1.1 Dynamic scene update.

Performing such a scene update for every small window variation is not efficient, so measuring discrete steps and updating the window in each step is more convenient

for management. In Figure 6.1.1, each small rectangular patch represents a terrain block or terrain tile. The window which includes the visible terrain tiles is called the scene map, and window changes cause an update for this scene map at every discrete step. During this update the new patches are loaded and old patches are freed from the memory. If terrain patches are in equal resolution for every area of the scene map such a diagonal window change as in Figure 6.1.1 will cause us to load 13 terrain patches, which is the maximum change difference, and horizontal change will cause 7 patches to be loaded, which is the minimum change difference. If window is of size $n \times n$ grids, maximum load count is $2n-1$, and minimum load count is n. For very large n this loading process will become a bottleneck and updating delays will occur, also using same resolution of terrains for each patch will increase the triangle count and decrease the frame rate. Pajarola solved this problem sampling each terrain patch according to the camera position and direction and displayed them with his rendering algorithm. Such a sampling creates different levels of details for each tile with different resolution as in Figure 6.1.2.

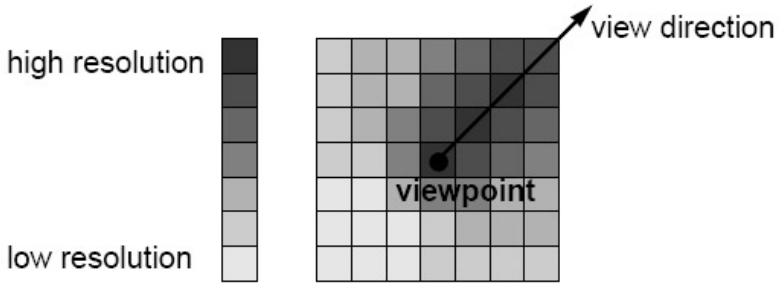


Fig. 6.1.2 Level-of-detail distribution.

In this method a terrain patch is loaded that is visible for the first time exactly at the specified level-of-detail and it is updated incrementally as required by changes of view coordinates up to the full resolution if necessary. Unused heightfield data is never deleted until the complete patch is thrown out of the scene map. Even using sampling for level-of-detail paging, complete $n \times n$ terrain patches will always be in the memory and will cause memory overflow as n increases. This method is also convenient for flight simulators not for general purpose, because random orientation changes will cause an up to now refinement of terrain patch sampling.

M. Reddy and Y. Leclerc used a quadtree based representation for paging in [5]. They progressively downsampled the heightmap to produce a multi-resolution pyramid.

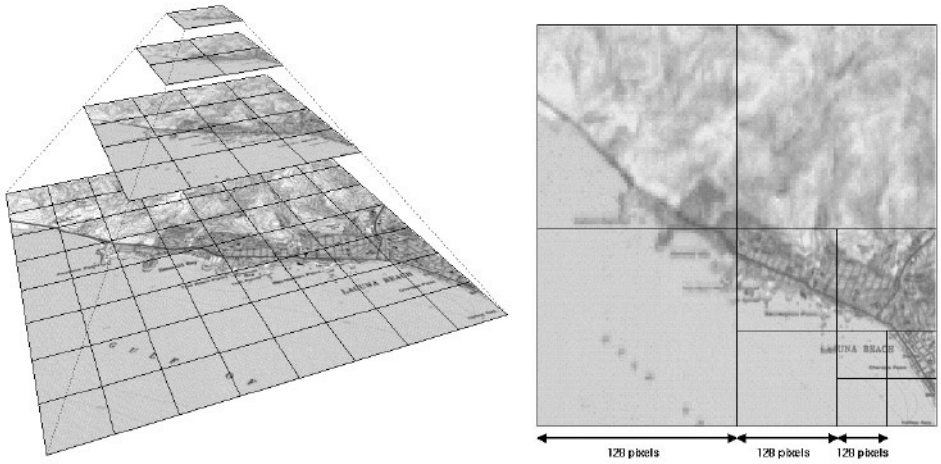


Fig. 6.1.3 Tiled pyramid and quadtree representation.

In Figure 6.1.3, left image shows four different resolutions of a digital map where each level is segmented into regular grid of patches, and right image shows the quadtree representation for resolution distribution of regions. Patches of each detail level has the same number of data points. Scene map should be updated as windowing method in [13], and a new quadtree structure must be created for every discrete step. Here window size should be a power of 2 to manipulate the quadtree structure for a scene map. It is effective to use such a structure for memory management especially in a restricted memory conditions. In order to create such a structure we should sample the heightfield for each level-of-detail on run time or previously. Run time sampling may reduce the performance of rendering so preprocessing is preferable, but the advantage of run time sampling is to reduce the load of external storage, because at preprocessing step for each block additional level-of-detail data is created and stored in the storage device of the computer. Using this technique, for example, if we want to represent a heightfield of $64K$ size with four levels-of-detail as in Figure 6.1.3, we will have to use an external storage of $64K + 16K + 4K + 1K = 85K$, nearly 33% more than the actual data. This storage increase is desirable against the performance decrease when such a preprocessing is

not implemented. One drawback of this method occurs on level change boundaries, we will explain this problem and our method to solve it in the next section in detail.

6.2 Paging Method

The paging algorithm in [5] is more convenient for very large scaled terrain rendering as we stated before. Because managing terrain tiles with quadtrees will gain us much more memory and processing time than one by one tile rendering. External storage requirement of this algorithm is also acceptable for us. Consider that we want to use n levels for detail management and each tile has an equal size of x KB in hard disc. We can create a scene map of 4^{n-1} size and the size of this scene map without level management on disc is $4^{n-1}x$ KB. While with level management its size may be calculated as in Equation (6.1).

$$size = \sum_{i=0}^{n-1} 4^i x \text{ KB} \qquad (6.1)$$

To emphasize the drawback of simple quadtree management in [5], let us have a scene map of size 8×8 as shown in Figure 6.2.1.

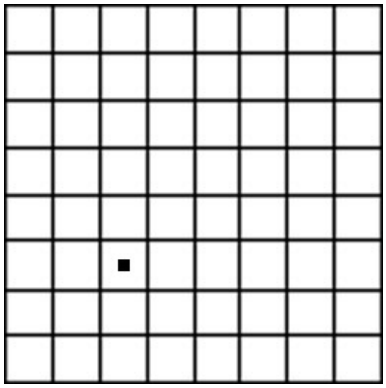


Fig. 6.2.1 A scene map of 64 tiles.

The black square in Figure 6.2.1 indicates the eye position, and according to this eye position the regarding quadtree structure will create a tile arrangement as in Figure 6.2.2.

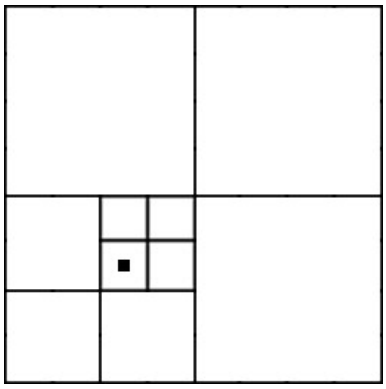


Fig. 6.2.2 Quadtree based tile arrangement.

In the Figure above if we look through up, right or up – right again we will see a terrain tile of same level with our current eye point terrain level, but if we look left, down or down – left we will see one level coarser terrain. This may disturb the observer when camera comes close to the boundaries of these coarser patches. If the camera altitude is high two level coarser terrain tile will also disturb the viewer. One way to decrease the effect of these undesirable level changes is to mark left, right, up and down and create a quadtree structure according to this arrangement. The result is shown in Figure 6.2.3.

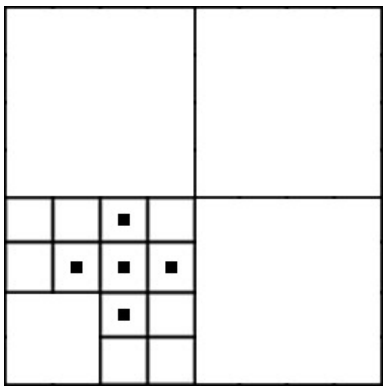


Fig. 6.2.3 Additionally marked tile arrangement.

In Figure 6.2.3 up, right, left, down, up – left, up – right and down – right tiles are of the same level but down – left tile is of one level coarser and it is so close to the viewer position. Changing the marking positions we may force all 8 neighbor tiles to be the same level of detail as in Figure 6.2.4.

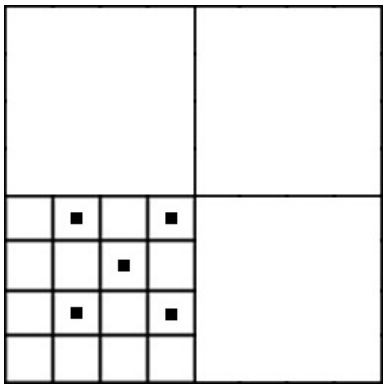
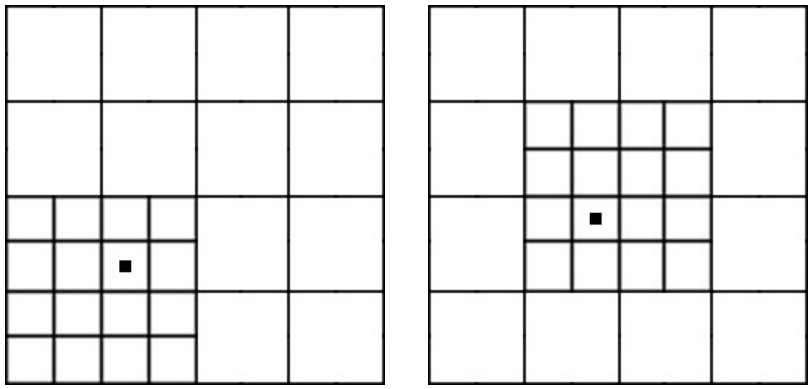


Fig. 6.2.4 Changing marking positions.

In Figure 6.2.4 all neighbor tiles are forced to be the same level and this gives a desirable landscape view for low altitude flights, but at high altitude flights the level difference of two will be obvious and disturb the viewer. So there should not be much level difference when the viewer is close to a boundary point. The desirable and possible quadtree arrangement of the scene map in the figures above is given in Figure 6.2.5.



(a) (b)

Fig. 6.2.5 Desirable and possible scene map arrangement.

The viewer position in Figure 6.2.5.a is moved to its up – right position and new quadtree arrangement is created in Figure 6.2.5.b. In Figure 6.2.1 we had 64 terrain tiles that have to be loaded and displayed, in Figure 6.2.2 using a simple quadtree arrangement we obtain 10 terrain tiles but with a visually problematic arrangement, and in Figure 6.2.5 our tile number is 28 but with a good tile arrangement. As we will indicate in experimental results section, loading and rendering 64 terrain tiles

causes both memory and GPU restriction artifacts. Instead loading 30 tiles is more convenient for visualization.

In order to create such a structure we can merge different quadtrees with different levels. Let us have a scene map of $16 \times 16 = 256$ terrain tiles, so we will have 5 level-of-detail terrain patches. Figure 6.2.6 shows its possible quadtree arrangement.

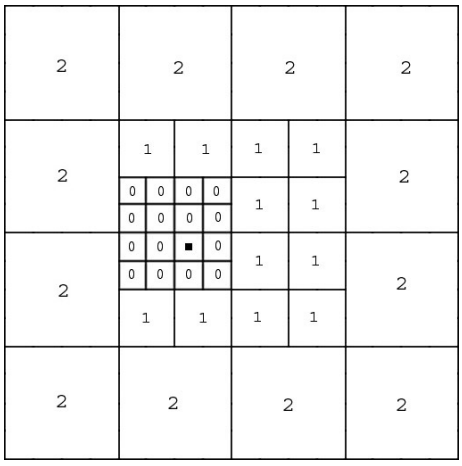
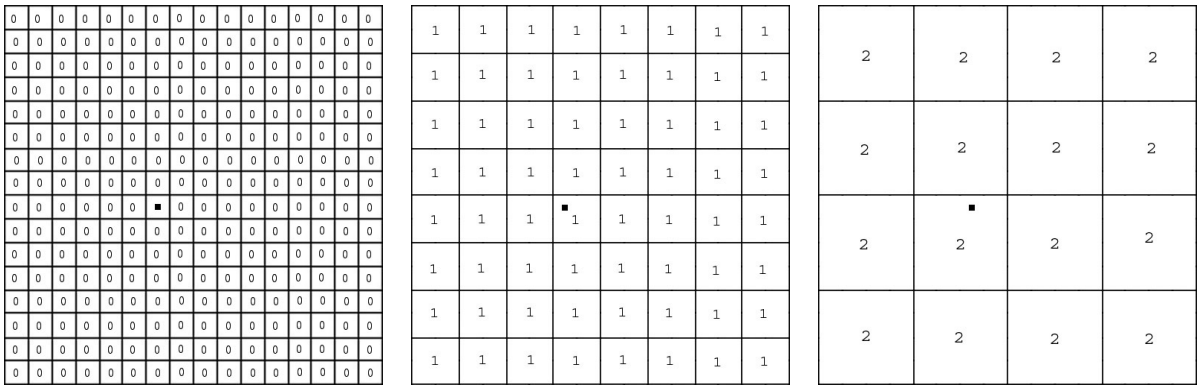


Fig. 6.2.6 Terrain tile arrangement for a scene map of 256 tiles.

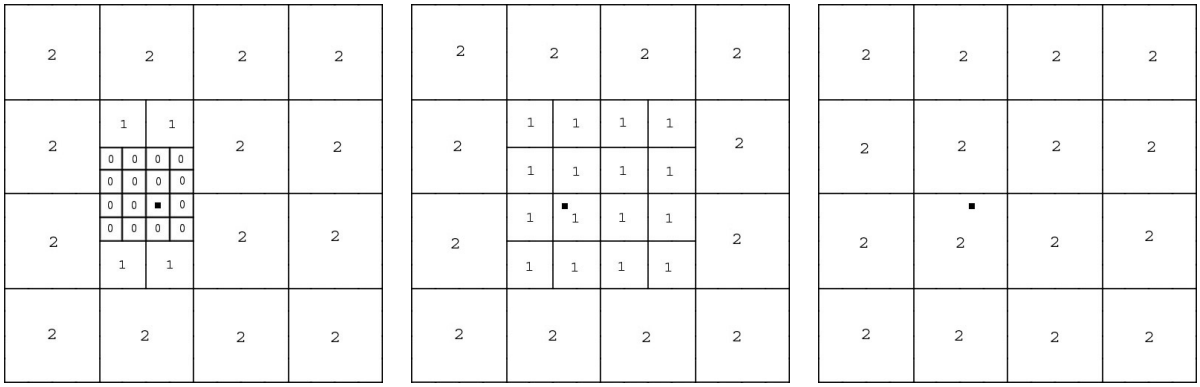
Instead of having 5 levels-of-detail, in Figure 6.2.6 we use 3 levels in order to keep the level difference restricted. Creating such a structure requires 3 different quadtree refinements for each level of detail.



(a) (b) (c)

Fig. 6.2.7 Three different levels used for restricted scene map creation.

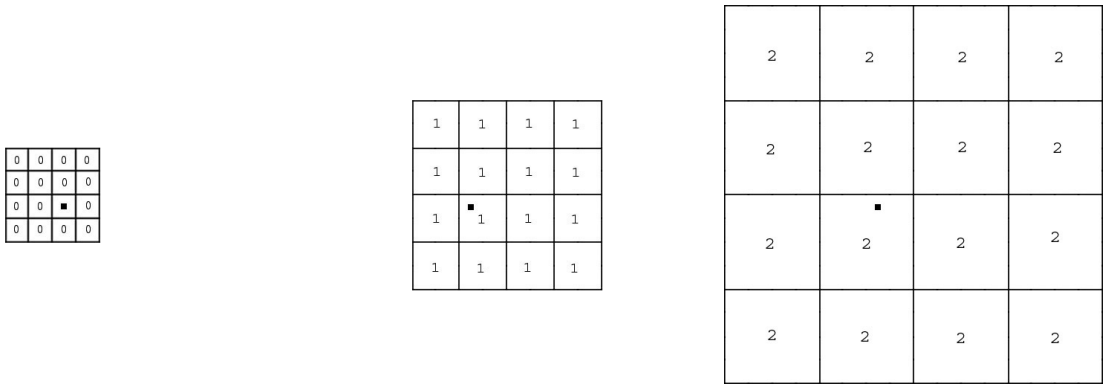
Refining each level by marking as in Figure 6.2.4 we obtain the following quadtree refinements.



(a) (b) (c)

Fig. 6.2.8 Quadtree refinements for each level.

In Figure 6.2.8 (a) the most detailed tile is marked with 0, in (b) with 1 and in (c) with 2. To create such a structure as in Figure 6.2.6 we choose these most detailed tiles from each refined level as in Figure 6.2.9.



(a) (b) (c)

Fig. 6.2.9 Chosen most detailed parts of each quadtree refinement.

Merging the refined and chosen detailed parts in Figure 6.2.9 with a priority of detail we obtain the desired and possible refinement for scene map. According to the flight altitude another refinement can be done. In this refinement the level number is decreased and the steps above are repeated for the new and coarser levels-of-detail.

7 TERRAIN TEXTURING

7.1 Overview

Rendering heightmaps only displays the geometry of a terrain. In order to create a more realistic terrain we have to paint its surface with a proper picture. Texture mapping is such a technique where we can use to bind a two dimensional texture on a three dimensional terrain. Texture mapping is hardware assisted painting method where due to GPU usage only thing to do is to find the proper texture coordinates for corresponding vertices. Texture coordinates are simply called *uv* coordinates and vary between a range of $[0,1]$. To stretch a single texture across the landscape we have to make every vertex in the landscape fall within this range.

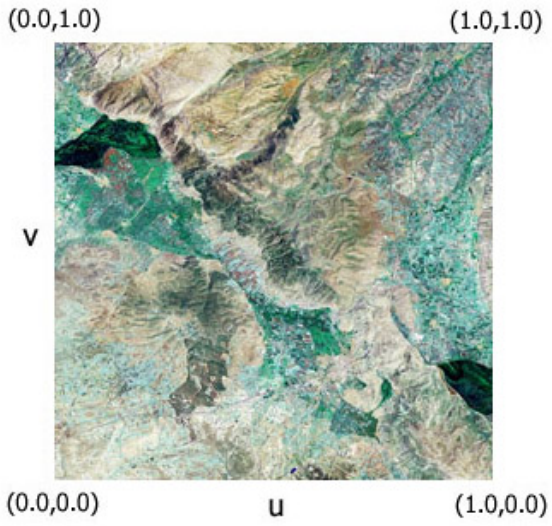
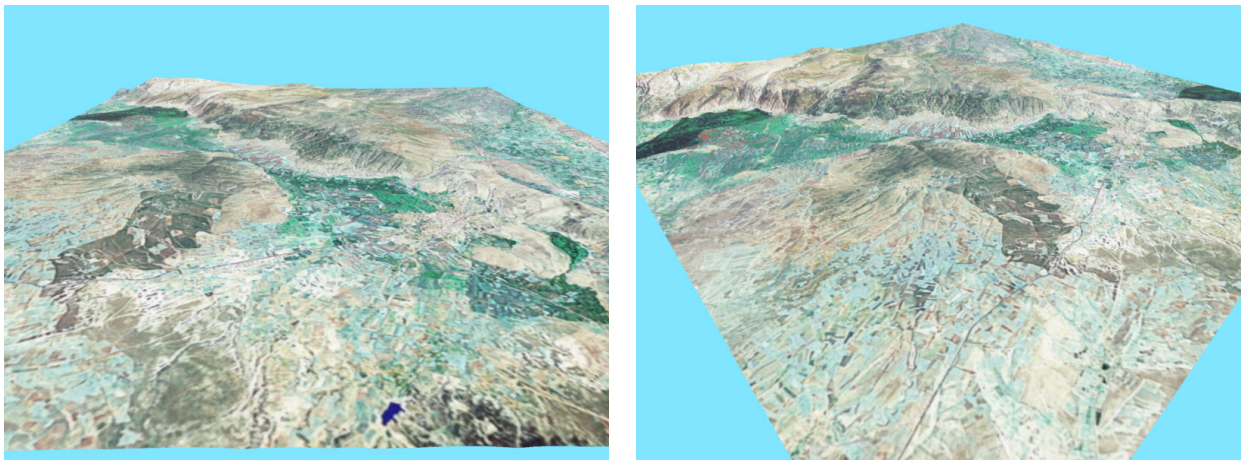


Fig. 7.1.1 Stretching a single texture.

In Figure 7.1.1, we can see a texture of a landscape area. South west corner of this texture gets texture coordinates of $u = 0.0$ and $v = 0.0$. As we obtain vertices from west to east the u coordinate will increase and reach to a value of $u = 1.0$. This is valid for v also, and at the north east corner the texture coordinates will be $u = 1.0$ and $v = 1.0$. In Figure 7.1.2 we can see the texture in Figure 7.1.1 which is binded to a heightmap.



(a) View from south (b) View from south west

Fig. 7.1.2 Texture binded to a heightmap.

Memory usage of textures depends on the graphics hardware memory. Before binding operation the texture is read from file and loaded to main memory. Then it is uploaded to the texture memory of GPU and then cleared from the main memory. By this way main memory is only used as a swapping area, so loading a texture does not increase our main memory usage, it only increases the GPU memory usage. As a result we can load as much as textures for our terrain according to our graphics hardware.

7.2 Large Textures

As we stated before, texture is a two dimensional image. The quality of this image decides the quality of our texture. Quality is not only simply the resolution of our texture but also ratio of texture pixels to screen pixels. Imagine a terrain that completely fills the camera view. If a small texture is applied to this geometry the result will be a blurry texture on terrain. If the terrain fills a small area on the camera view blurring may not occur. The higher the resolution of the texture, the less blurry the result, in other words as the ratio of texture pixels to screen pixels grow quality increases.

Texture in Figure 7.1.1 is an 2048×2048 image and it is applied to a 27 km^2 area. If we have a closer look at this terrain it is obvious that the image will be blurred, because nearly one pixel images will be applied to 13 m^2 areas.

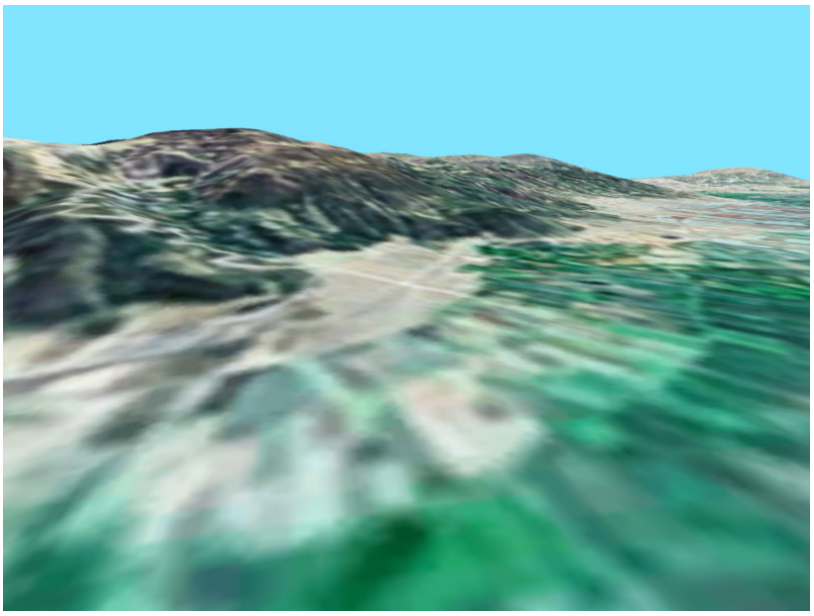
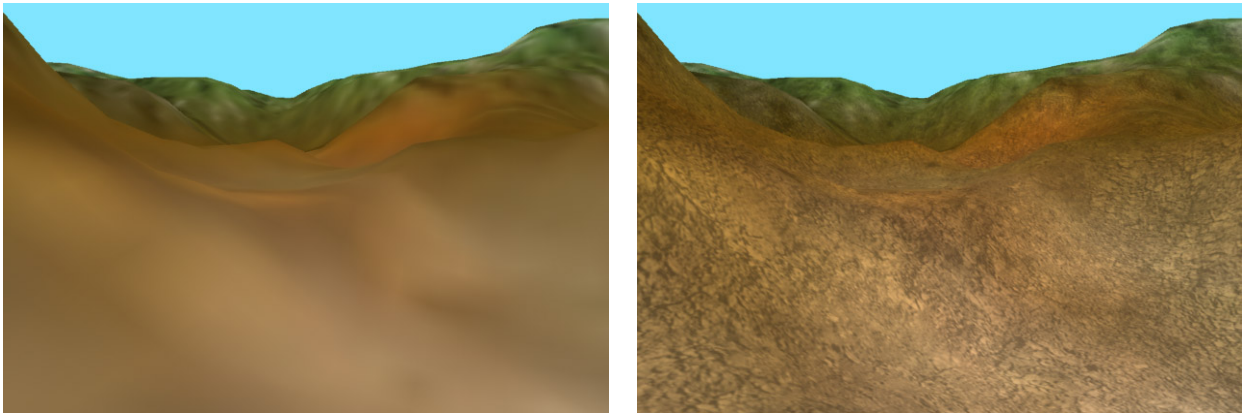


Fig. 7.2.1 A closer view of terrain.

In the perspective close view of terrain in Figure 7.2.1 closer areas are highly blurred. As we look farther from the camera the blurring decreases and visual quality increases because of texture pixel screen pixel ratio.

One method which is suggested in [16] is to apply detail map to obtain more realistic, not blurred terrains. A detail map is a grayscale texture that is repeated many times over a landscape using multitexturing property of graphics hardware. With actual texture this detail map is also uploaded to GPU memory and binded to the terrain with the actual one. In Figure 7.2.2 we can see the effect of detail mapping on a terrain.



(a) Without detail mapping (b) With detail mapping

Fig. 7.2.2 Effect of detail mapping while texturing.

Detail mapping is an effective method for nature landscapes, because it adds cool nuances such as cracks, bumps, rocks and provides a more natural view. But this method cannot be applied to actual image desired applications such as Global Information Systems, because it adds artificial elements to texture where in GIS applications all elements have to be real.

7.3 Large Texture Management

Large scaled texture mapping problem is the same problem as we encountered while terrain rendering. In actual landscape modeling, heightmap of the terrain is merged with the aerial photograph of the landscape and the model is created. The resolution of aerial photographs are very large, in order to obtain a clear view from 300 meters above an area we must have a sampled photograph with at least 1 meter resolution. So to bind such a photograph on an area of 108 km^2, which represents a one degree area, we must have an image of size 108000×108000. Uploading such an image to the GPU memory is not possible with current graphics hardware. A 1024×1024 bitmap of 24 bits requires a RAM of 3 MB, a bitmap that covers a one degree area is nearly 33 GB. Current graphics hardware memory is maximum 256 MB and supports a maximum single texture size of 4096×4096, so a paging system is essential for management.

Our texture management and paging method is the same as we applied to heightmap management. Again textures will be leveled with a preprocessing step and will be stored in the external storage device. Level decision according to the position is done according to the extracted quadtree and difference levels are uploaded to the GPU memory. The following figure in Figure 7.3.1 is the terrain patch without detail texturing.

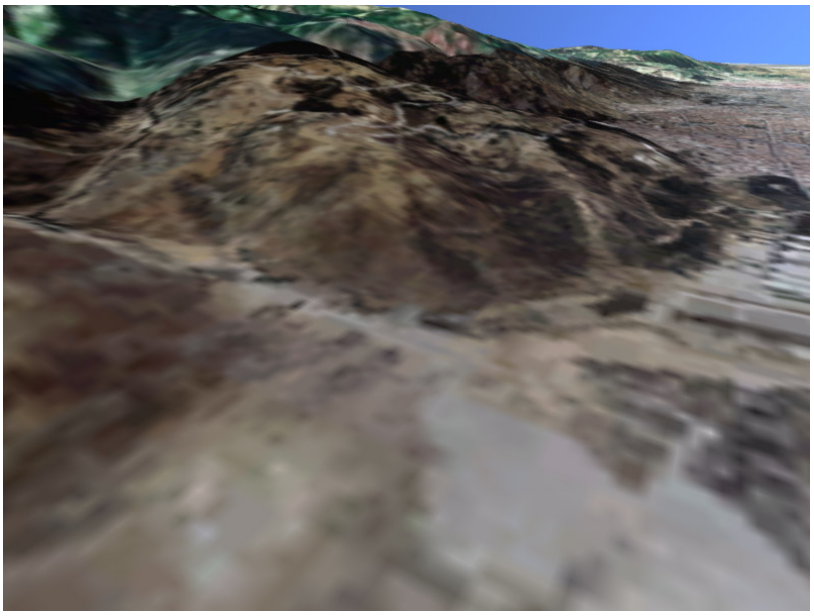


Fig. 7.3.1 Terrain without detail texturing.

After applying detail texturing we obtain the following textured terrain in Figure 7.3.2.

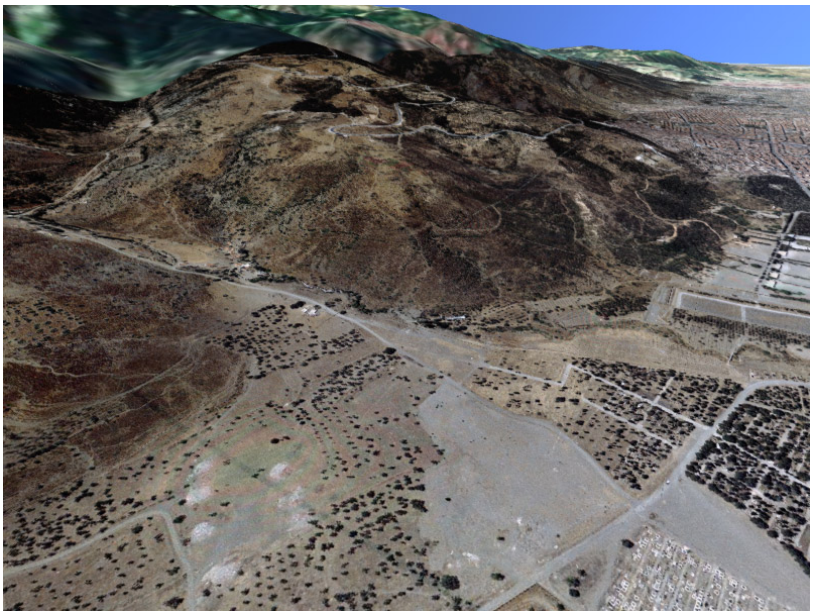


Fig. 7.3.2 Terrain with detail texturing.

The blurring effect in Figure 7.3.1 is disappeared in Figure 7.3.2. The texture used here is sampled with a resolution of 50 cm, downsampled to a depth of 6 levels and stored in the storage device. As we look farther away from the camera resolution

decreases but this resolution loss does not cause blurring due to the quadtree refinement of these textures and texture to screen pixel balance.

The resolution of our heightmap and texture may be different in many situations. They may be equal or texture resolution is greater. General situation is the second one in GIS applications. So more than one texture may be wanted to be stretched on a terrain tile. The problem here is to obtain the correct texture coordinates for considered terrain vertex. We send texture coordinates with terrain vertices to the rendering pipeline in the last step of rendering. At this step hardware uses the texture in his active texture buffer with the given *uv* coordinates and binds that part with the given vertices. Terrain rendering algorithms uses hardware assistance such as vertex arrays for fast rendering. Vertex array is a structure which contains the vertex positions of terrain triangles. Sending this structure to the pipeline with an appropriate texture coordinate array causes fast rendering of terrain. But vertex arrays can not be used if we intend to apply detail texturing, because hardware does not support more than one active texture buffering. So we will not be able to let the hardware know which texture corresponds to which texture coordinate. Using triangle fans may get rid of this behavior. The rendering a simple quad of a terrain is given in Table 7.3.1.

Table 7.3.1 Rendering a basic quadtree fan.

```
1     glBegin(GL_TRIANGLE_FAN);
2          glVertex(x, z);
3          glVertex(x – edgeOffset, z - edgeOffset);
4          glVertex(x, z – edgeOffset);
5          glVertex(x + edgeOffset, z – edgeOffset);
6          glVertex(x + edgeOffset, z);
7          glVertex(x + edgeOffset, z + edgeOffset);
8          glVertex(x, z + edgeOffset);
9          glVertex(x - edgeOffset, z + edgeOffset);
10         glVertex(x - edgeOffset, z);
11         glVertex(x - edgeOffset, z - edgeOffset);
12    glEnd();
```

If we had bind a single texture to this terrain only one binding would be enough before rendering. As we want to extract each texture and its coordinate for detail texturing we have to test this before each triangle fan rendering as in Table 7.3.2 bind this triangle fan with the corresponding texture.

Table 7.3.2 Rendering a basic quadtree fan with appropriate texture.

```
 1     TestDetailTexture();
 2     glBegin(GL_TRIANGLE_FAN);
 3          glVertex(x, z);
 4          glVertex(x – edgeOffset, z - edgeOffset);
 5          glVertex(x, z – edgeOffset);
 6          glVertex(x + edgeOffset, z – edgeOffset);
 7          glVertex(x + edgeOffset, z);
 8          glVertex(x + edgeOffset, z + edgeOffset);
 9          glVertex(x, z + edgeOffset);
10          glVertex(x - edgeOffset, z + edgeOffset);
11          glVertex(x - edgeOffset, z);
12          glVertex(x - edgeOffset, z - edgeOffset);
13     glEnd();
```

Doing such a test before rendering may seem to decrease the performance, but this test is nothing more than checking some indices and binding the proper texture to the heightmap.

8 EXPERIMENTAL RESULTS

8.1 Overview

In this section we will examine the results of some tests and the final product of our work. The computer configuration used in these experiments is given below:

- Processor: Intel Pentium 4 CPU 3.20 GHz with Hyper Threading (HT) technology
- Storage Device: Two hard discs each with 80 GB capacity
- Main Memory: 2.00 GB
- Graphics Hardware: NVIDIA GeForce FX 5900 Ultra
- GPU Texture Memory: 256 MB

The tests below are done in order to choose the proper triangulation and paging algorithm for large scaled terrain visualization. According to the test a complete system is designed and implemented.

8.2 Loading Time Test

In this test we measured the loading time of our terrain algorithm and the algorithm of Röttger in [6]. Repeatedly terrain blocks are loaded and freed from the memory and the time is measured against the block count. The regarding performance chart is given in Figure 8.2.1.

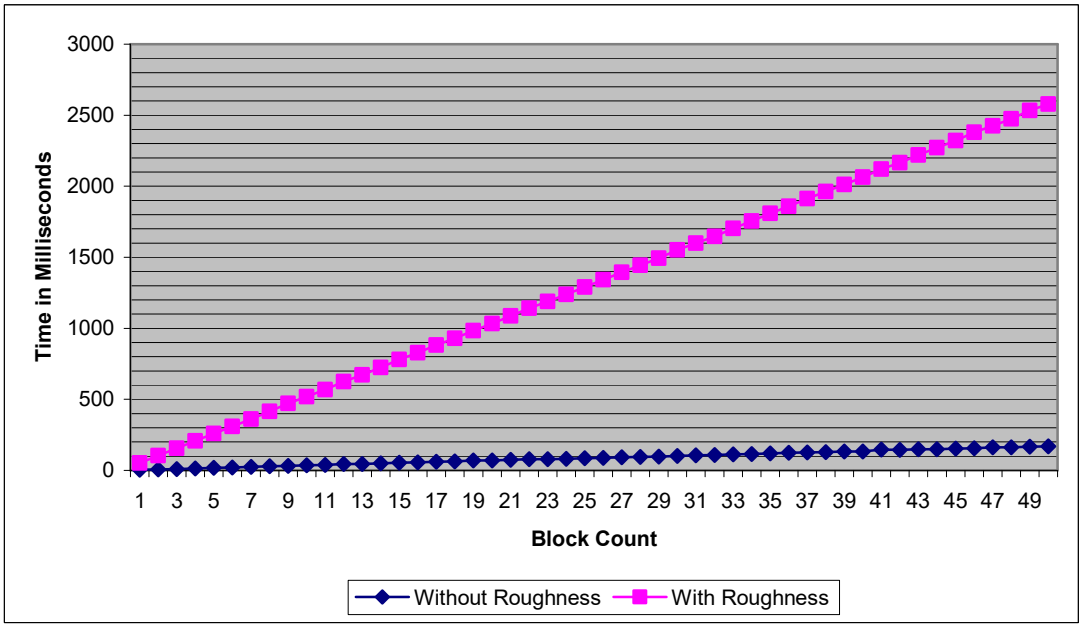


Fig. 8.2.1 Block loading times for number of blocks.

In [6], Röttger suggested a second criterion to increase the resolution for regions of high surface roughness. But this requires an extra work before rendering. In Figure 8.2.1, we can see the result of doing such a calculation. Loading 50 blocks on run time without any roughness algorithm requires 170 milliseconds and with roughness algorithm requires 2578 milliseconds, so roughness calculation brings nearly 15 times of processing time cost.

Loading time is not very important if we fly over the terrain slowly regarding the block size. For example, if one block flight time is 60 seconds we can easily load much more than 50 blocks until we cross to the next block. But when our flight speed is fast relative to block size loading time will be a bottleneck and desired blocks would not be loaded on time. There is one important point that we should take into account event if our block size is large relative to our flight speed. Block loading may be done in two ways, discrete or continuous loading. In discrete loading first navigation is paused, blocks are loaded and then navigation is again started. Navigation pause time is the same time for loading (Figure 8.2.2).

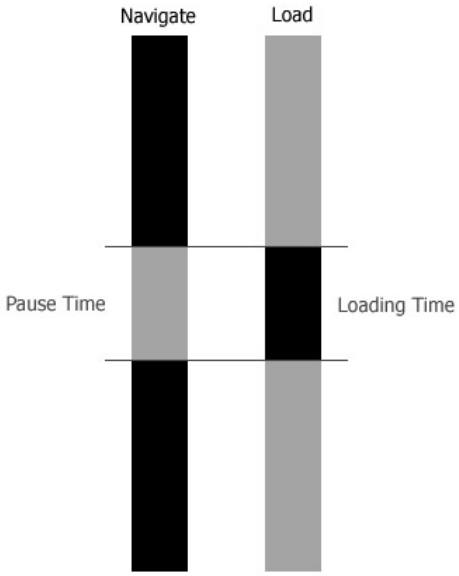


Fig. 8.2.2 Discrete loading process.

In continuous loading navigation is never paused, loading is done in background and scene is updated when loading ends (Figure 8.2.3).

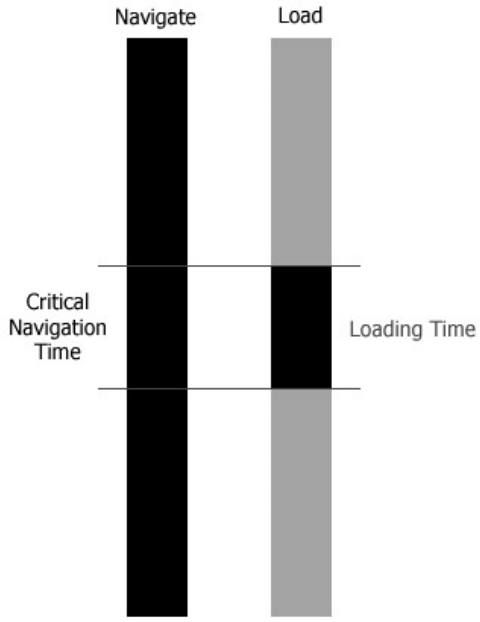


Fig. 8.2.3 Continuous loading process.

In Figure 8.2.3, critical navigation time is the time slice when both loading and navigation is processed. In order not to stop navigation, another thread is created and blocks are loaded from this thread. During loading terrain is rendered from the main

thread of the process. Creating such a child thread causes the process to decrease the run time of the main thread and give some time to child thread. Such a thread balancing of the operating system causes the frame rate to decrease. Even if we try to organize the thread priority of the process such a frame rate loss is inevitable. As a result decreasing the loading time may decrease this frame rate loss effect.

8.3 Horizontal and Vertical View Test

In this test we randomly navigated on terrain for nearly 20 second with different viewing angles. During navigation we measured the frame rate for each 5 frames and collected more than 1000 samples. The chart regarding frames per second and angle deviation is given below.

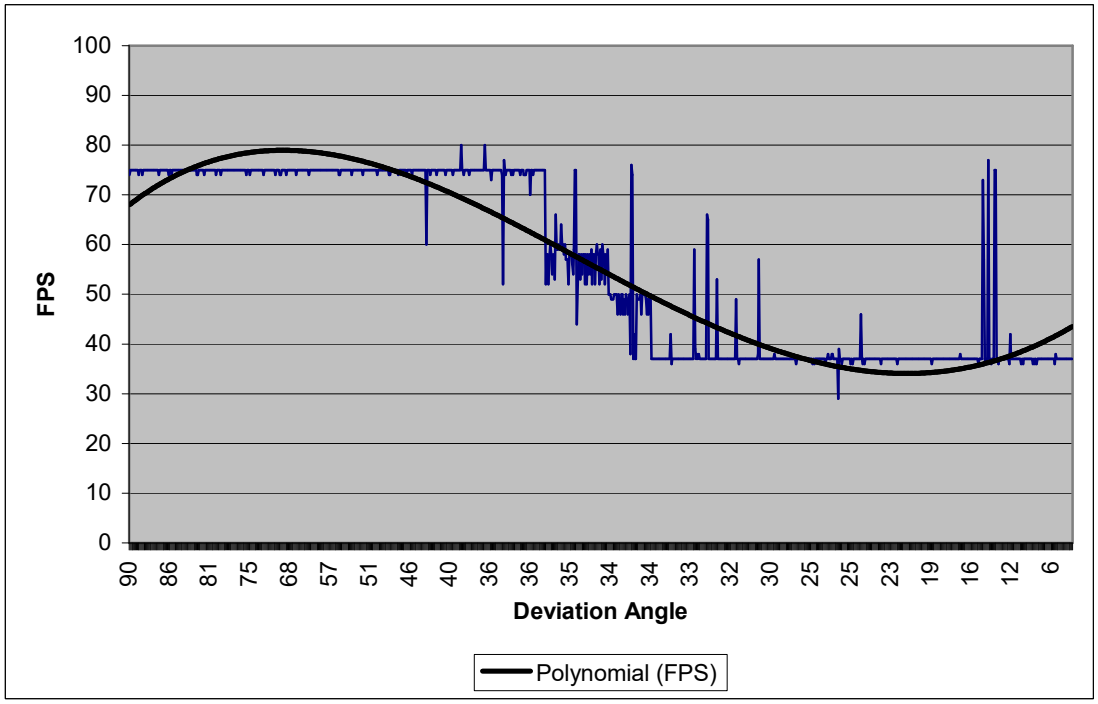


Fig. 8.3.1 Sampled FPS regarding deviation angle.

In Figure 8.3.1, *x* axis represents the deviation angles from horizontal look-at vector. 90 degrees is the most vertical look to the terrain and the 0 degree represents a parallel look. Starting from 90 degrees until 36 degrees of deviation we are able to acquire a frame rate 75 FPS. If we start to decrease the deviation from 36 degrees to 0 our frame rate is decreased and in the worst case a frame rate of 37 is obtained.

The smooth curve on the figure represents the frame rate change with a fourth degree of polynomial. The transition band of the curve is between 23 and 64 degrees, and this is the angle slice where frame rate changes. This change is not linearly increasing or decreasing and it depends on a threshold where below it the frame rate is completely decreased and above it the frame rate is completely increased. The threshold depends on the altitude of our camera and increases as altitude increases. In Figure 8.3.1, this threshold is 35 degrees where below this point the complete horizontal scene is appeared in the view frustum and rendered. That is why the frame rate is decreased suddenly.

8.4 Roughness Effect Test in Horizontal and Vertical Views

This test is the same test as in section 8.3, but this time the surface roughness is taken into account. The chart regarding frames per second and angle deviation is given below.

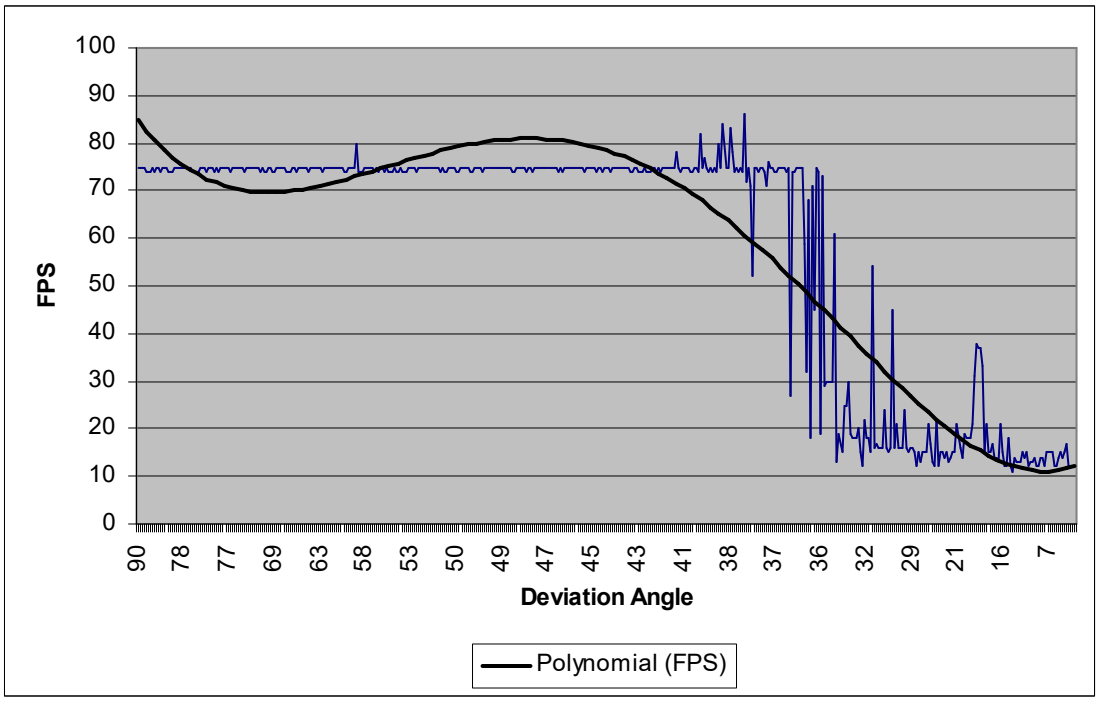


Fig. 8.4.1 Sampled FPS regarding deviation angle with roughness effect.

In Figure 8.4.1, we obtained a frame rate of 75 FPS while we look vertically and 13 FPS while we look horizontally. Rough areas are generally heights like mountains,

and due to roughness calculation more triangles are attached to these heights. A horizontal look to these areas causes the graphics hardware to render all these triangles and to decrease the frame rate down to 13 FPS. The transition band of the polynomial in Figure 8.4.1 is between 10 and 48 degrees, and the threshold is 36 degrees. Threshold is not changed because our altitude is same as in previous test.

To compensate the effect of roughness we can use the formula in (5.9), where θ is our deviation angle. The regarding analysis chart is in Figure 8.4.2.

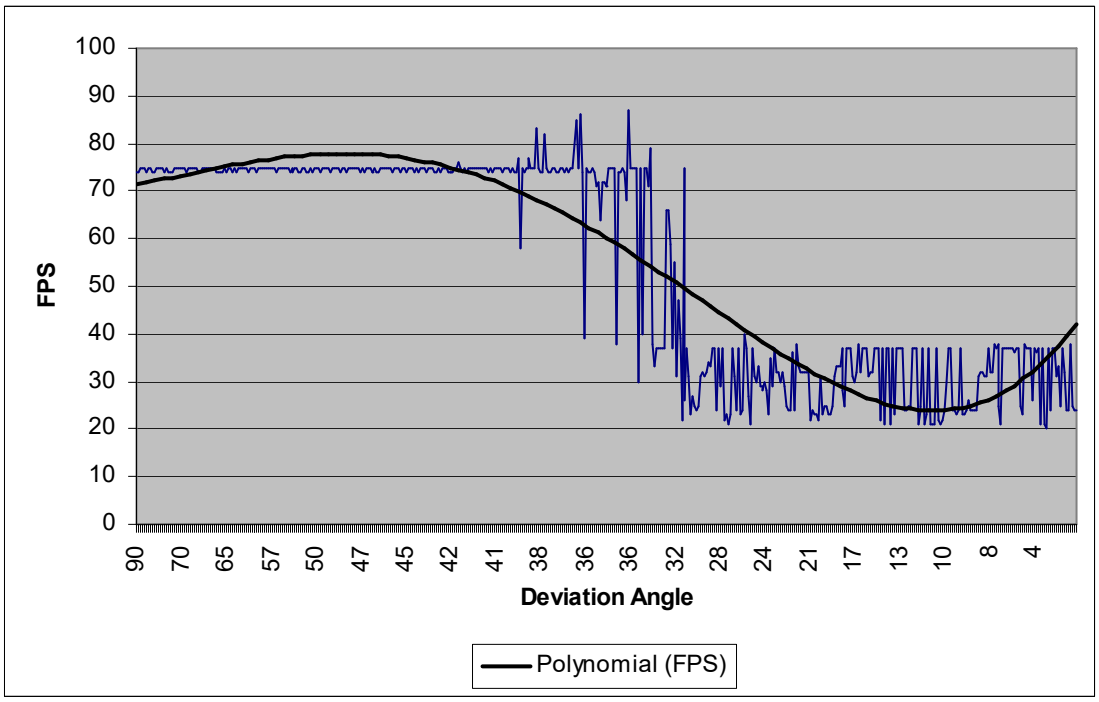


Fig. 8.4.2 Sampled FPS regarding deviation angle with roughness effect and angle compensation.

In the figure above the worst case frame rate is increased from 13 FPS to 30 FPS, but unstable ripples are created around this frame rate. Even a little camera angle movement affects the current triangulation and causes unstable frame rates. It also causes the popping effects to be more visible.

As a result of the experiments in Sections 8.2, 8.3 and 8.4 not using a roughness calculation in our terrain rendering model will increase the performance of the algorithm and will cause it to be more proper to real time applications.

8.5 Scene Map Organization Tests

In these tests we measured the visual quality result of created scene maps. Our first test is quadtree based tile arrangement quality test as in Figure 6.2.2. Our view position is given below in Figure 8.5.1.

Fig. 8.5.1 Scene map structure and camera orientation for single marked quadtree.

The Figure above is taken from an altitude of 100000 meters in order to see the large amount of the scene map and the camera orientation. Camera orientation is indicated with the small plane figure where here its look-at vector is through the north – west part of the image. The spaces between hierarchical levels are created on purpose to represent the structure even better. The left part of the scene map is empty, because the hierarch level desired for that part is not created. The following Figure is the screenshot of the landscape when it is viewed from the camera.

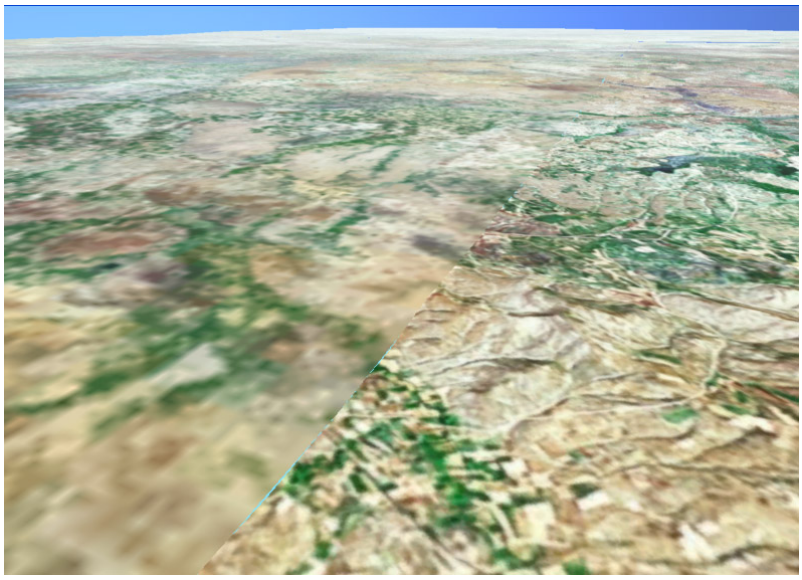


Fig. 8.5.2 Camera shot of the viewer in Figure 8.5.1.

In Figure 8.5.2, left part of the landscape is 3 levels coarser than the current level where the camera is located. As a result of such a scene map organization left part of the terrain contains blurred texture and low detail. High detailed vertices do not coincide with the low detailed ones properly and some cracking effect occurs close to the viewer. Marking scene map '+' as in Figure 6.2.3 creates the following scene arrangement.

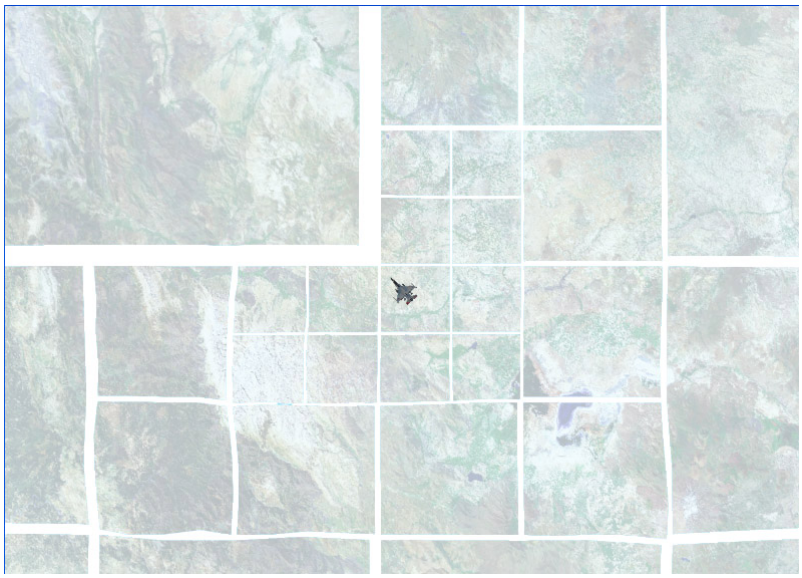


Fig. 8.5.3 Scene map structure and camera orientation for '+' marked quadtree.

In Figure 8.5.3, the left part of the scene map is also loaded. The camera orientation is through the north – west of the image and the regarding viewer screenshot is in Figure 8.5.4.

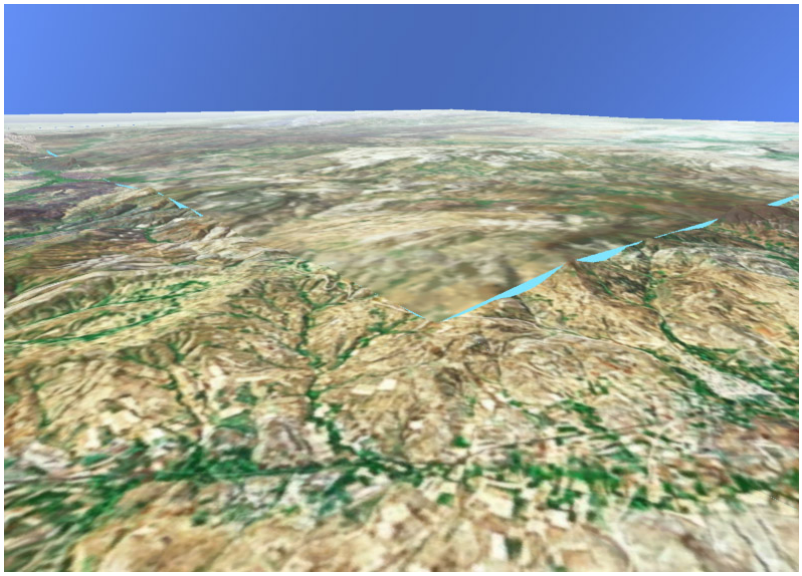


Fig. 8.5.4 Camera shot of the viewer in Figure 8.5.3.

In Figure 8.5.3, even a coarser level is loaded for north – west part of the scene map, and this causes undesirable visual effects as in Figure 8.5.4. Cracks are more obvious here due to level difference. Marking scene map '×' as in Figure 6.2.4 creates the following scene arrangement.

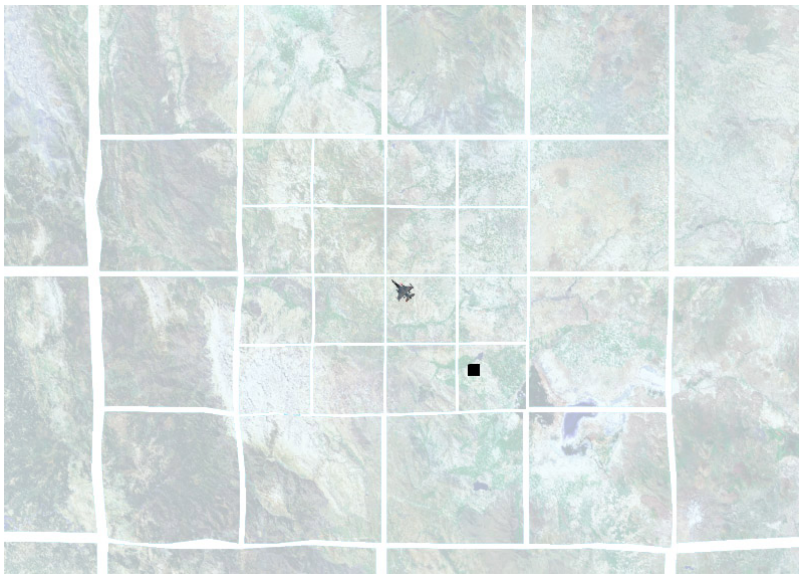


Fig. 8.5.5 Scene map structure and camera orientation for '×' marked quadtree

In Figure 8.5.5, all blocks are properly loaded, also level hierarch is good. The viewer screenshot from the camera position and orientation is given below.

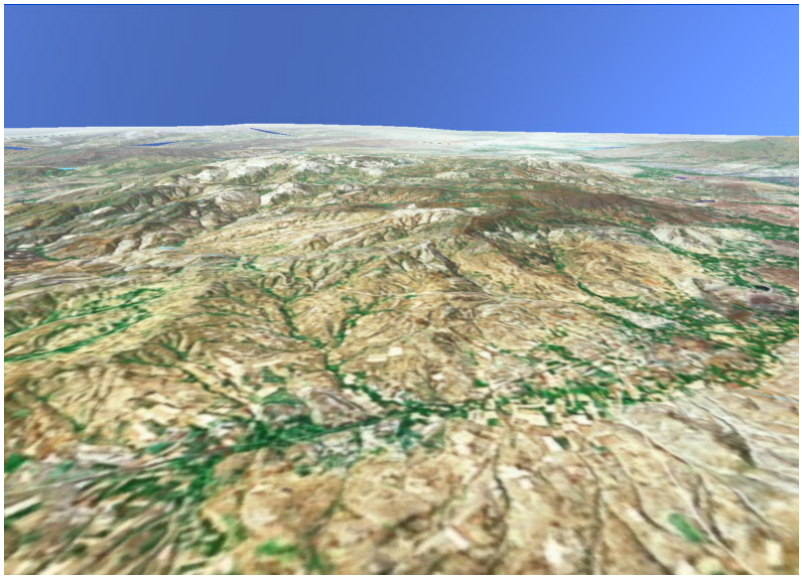


Fig. 8.5.6 Camera shot of the viewer in Figure 8.5.5.

In Figure 8.5.6, problems caused by level difference in Figure 8.5.4 are fixed and closer cracks are prevented. If we move the camera to the position of the black square in Figure 8.5.5, we obtain the following scene map.

Fig. 8.5.7 Scene map structure after moving the camera position.

As you can see from the Figure above, the high level difference of blocks caused to disappear the one larger block from the west side of the terrain. That is because we do not have that low leveled terrain block in our terrain hierarchy. Even if we had created that hierarch and rendered it using that level, there would be cracks due to high level difference and the textured image would be very blurred. Looking forward to north in high altitudes will also make low detailed terrain patches more noticeable. If we apply a quadtree hierarch as in Figure 6.2.6 we obtain the following Figure.

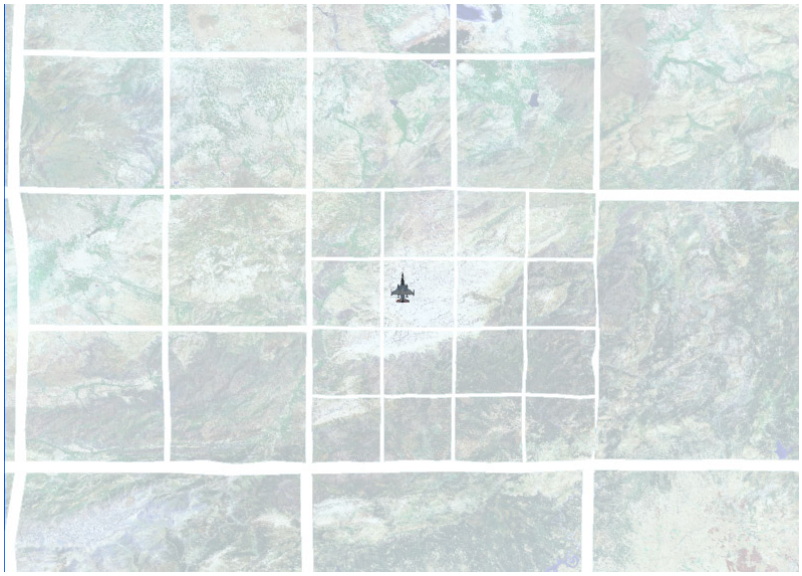


Fig. 8.5.8 Scene map structure after using a quadtree hierarchy as in Figure 6.2.6.

In Figure 8.5.8, the artifacts in Figure 8.5.7 are covered and more convenient structure for visual quality is obtained. Even if we increase the altitude of the eye point level changes does not disturb the viewer.

8.6 Loaded Block Count Test

In order to create a scene map without using the full resolution we have created and tested four different quadtree structures: normal quadtree, '+' marked quadtree, '×' marked quadtree and '×' marked restricted quadtree in the previous test section. In this section we randomly navigated on the scene map and calculated the loaded block count at each block change. During block change the organized scene map in the memory is compared with the updated scene map organization and difference blocks

are loaded, on the other hand blocks in the memory that are not included in the new scene map organization, are cleared and freed.

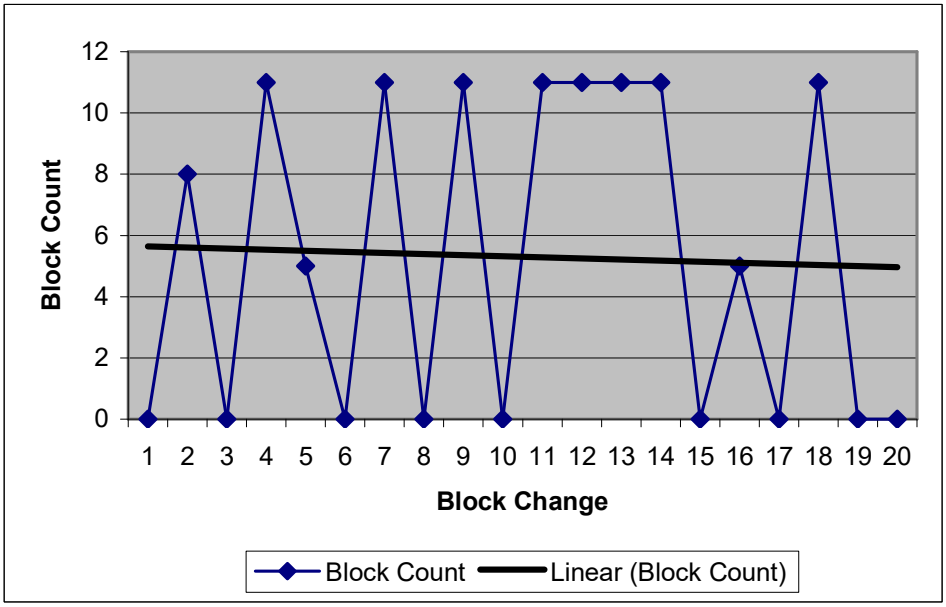


Fig. 8.6.1 Block count in normal quadtree.

In Figure 8.6.1, block count is measured against 20 block changes. In some changes block count does not change and remain 0 due to navigation between same levels of detailed blocks. But we observe a block count of 11 when we change our eye position from a high detailed level to a low detailed level and make that low detailed block to be subdivided, so we get oscillations.

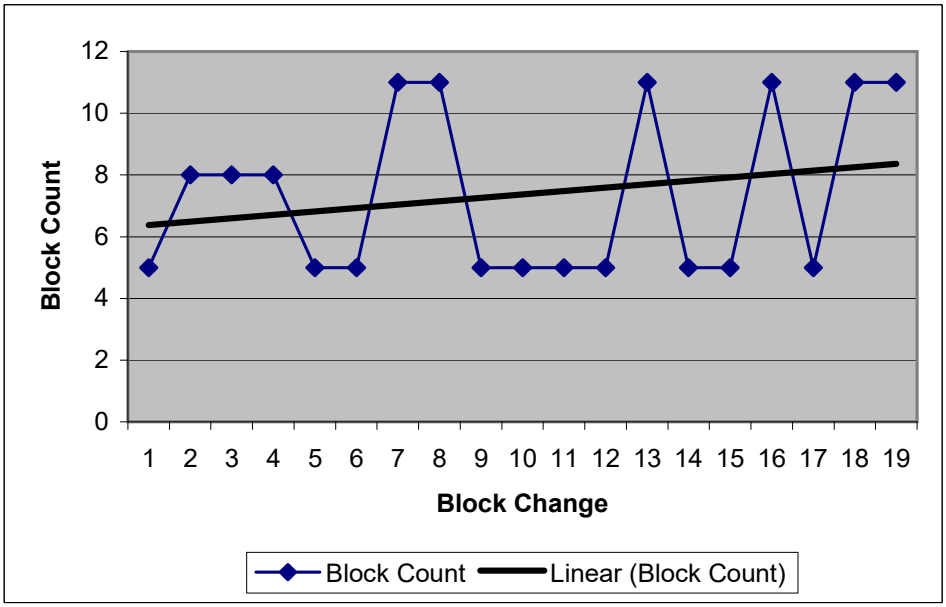


Fig. 8.6.2 Block count in '+' marked quadtree.

In Figure 8.6.2, minimum loaded block count is 4 due to the structure of '+' marked quadtree. Eye point changes from high level to low levels causes maximum 11 blocks to be loaded.

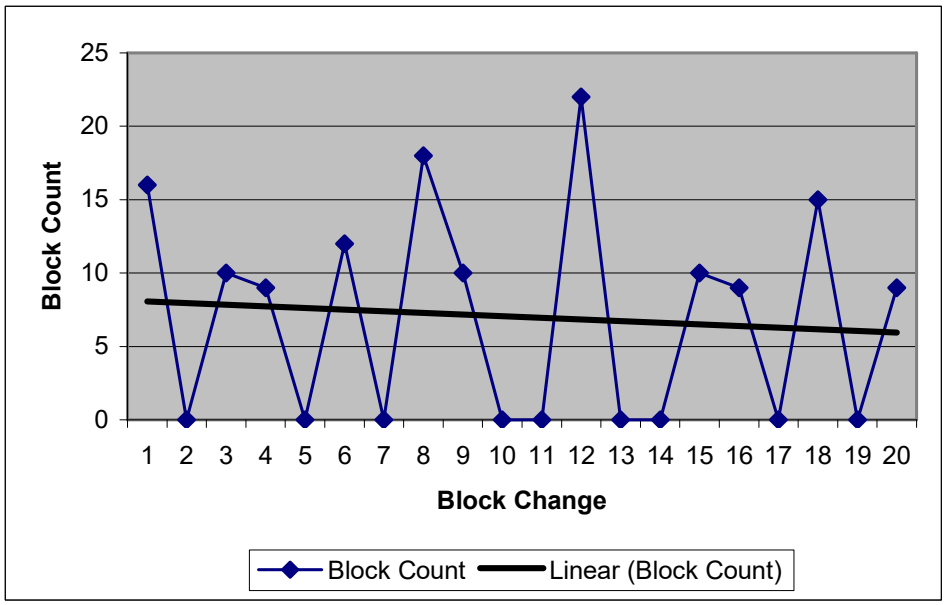


Fig. 8.6.3 Block count in '×' marked quadtree.

In Figure 8.6.3, sometimes we do not even need to load any blocks because position changes does not cause any difference in the scene map structure. But when we

change our position from a high leveled block to a very low leveled block we may obtain a block load count of 22.

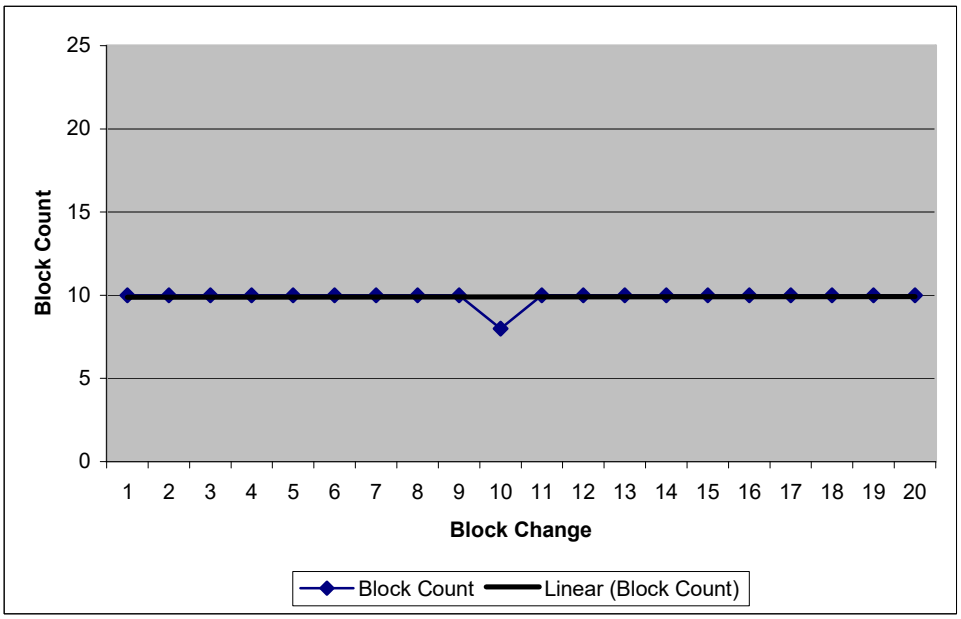


Fig. 8.6.4 Block count in '×' marked restricted quadtree.

In Figure 8.6.4, we obtained a generally stable block count while we navigate on terrain. Each block change causes some difference blocks to be loaded to the memory. Obtaining such a stable structure is important when we design our background loader. Considering a structure, our loader should be capable of loading the worst case of loaded block counts which are the maximum counts in each Figure above. The worst case scenario chart is given in Figure 8.6.5, where the maximum loaded block counts are displayed for each structure.

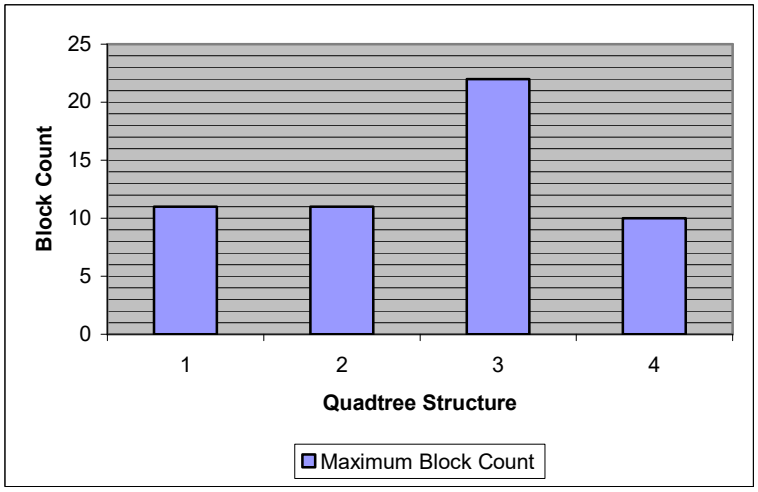


Fig. 8.6.5 Maximum block counts for each quadtree structure.

In the Figure above, our quadtree structure loads the minimum number of blocks in the worst case, so we will get the best loading performance and minimize the load time. From Figure 8.2.1, the load time needed for our structure is 35 ms.

The block counts above are difference block counts that are loaded due to eye position's block change. The total number of blocks in the memory during navigation is given below.

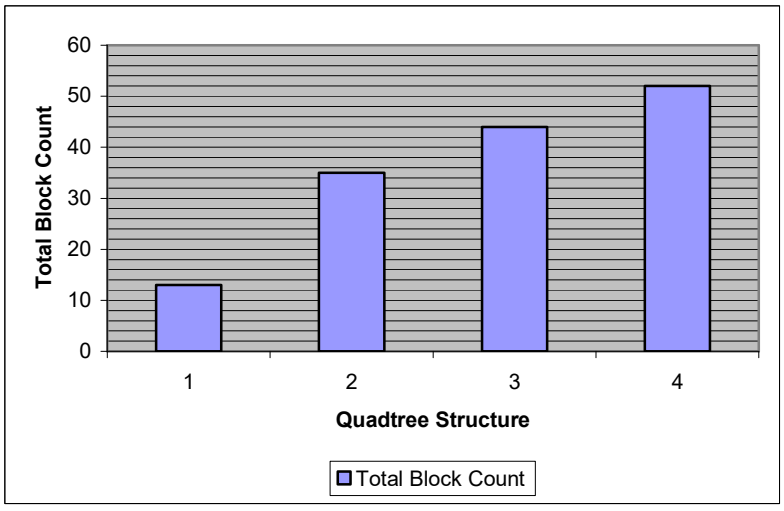


Fig. 8.6.6 Total block counts for each quadtree structure.

Total block count is vital when we have memory restrictions. In Figure 8.6.6, the regarding total block counts are 13, 35, 44 and 52. Now if we consider the first

initialization process of the complete terrain we should load 13, 35, 44 and 52 blocks for each regarding quadtree algorithm. During block change after initialization when we change block we will load extra 11, 11, 22 and 10 blocks respectively, so we will need a memory capable of restoring maximum of 24, 46, 66 and 62 blocks at the same time in the memory. If we do not have any memory restriction the performance of loading only depends on the loaded block count during run time, which is minimally 10 for our quadtree structure.

8.7 Implementation Details

During this thesis we created a generic and flexible terrain engine that can visualize every part of the earth in any resolution if heightmap data is provided. The coordinate system used in implementation is geographic coordinate system, where coordinates are related to the ellipsoid used to model the earth.

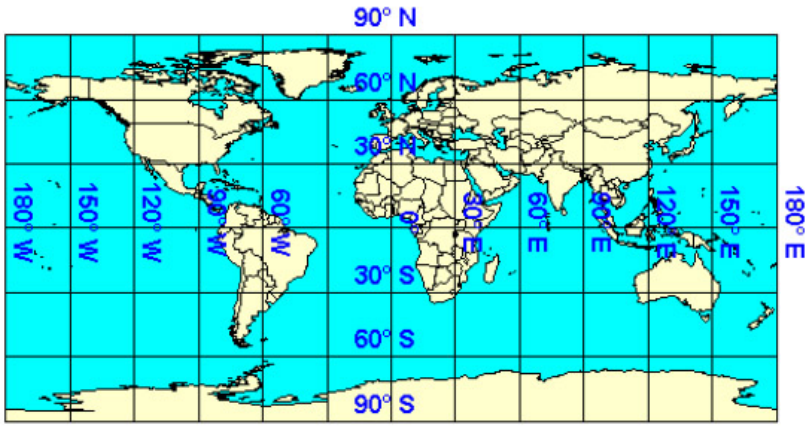


Fig. 8.7.1 Latitude/longitude graticule.

In Figure 8.7.1, latitude/longitude graticule of the earth is given. It is organized to locate points on the earth's surface via a grid of meridians and parallels. Each grid indicates a portion of 1 degree surface, which is a rectangular area of 108 km × 108 km. We are provided the elevation data of Turkey, which falls into a grid of latitude and longitude as in Figure 8.7.2.

Fig. 8.7.2 Latitude and longitude interval of Turkey.

In Figure 8.7.2, '**E**' indicates 'East' and '**N**' indicates 'North', where **E 000 – N 00** is the origin of the earth. Each 1 degree portion is subdivided into 8×8 sub-patches in the most detailed resolution, where each patch has 513×513 sample data points. Resolution organization of each 1 degree portion is given in Figure 8.7.3.

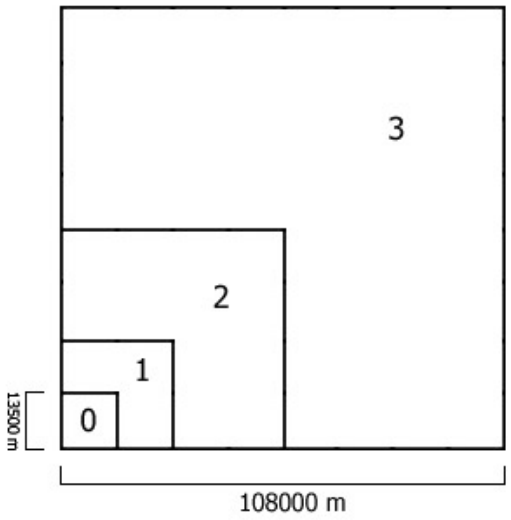


Fig. 8.7.3 Resolution organization of each 1 degree portion.

In high altitudes 108000 m is a reasonably proper viewing distance, taking this measure into account our scene map is of 324000 m × 324000 m size, which is equal to 24 × 24 highest resolution heightmap data. We used our rendering and paging mechanism to create and render such a scene map. Some parts are also detail textured with the same quadtree structure.

Background loading of both updated heightmap blocks and detail textures are run in different threads. To decrease the frame rate loss during this loading process the frame is locks to 60 FPS and additional time is used for loading. Even if we decrease our altitude and come closer to terrain, we are able to render and view the terrain in 60 FPS.

Textures are of 1024×1024 size and in Direct Draw Surface (DDS) format. The following table shows the RAM requirements for a BMP and DDS texture in megabytes.

Table 8.7.1 RAM requirements for BMP and DDS

Size of texture	24 bit BMP	DDS
1024x1024	3	0.5
2048x2048	12	2
4096x4096	48	8

As can be seen from Table 8.7.1, DDS is six times smaller than BMP and such a compression does not affect the visual quality of the texture image. We used detail textures of 0.5 m or 1 m in some areas of the terrain, so it is more convenient for us to use DDS format not to overflow the texture memory of the graphics card.

Some screenshots of our terrain engine is given in Appendix A.

9 CONCLUSION AND FUTURE WORK

In this thesis, a proper rendering and paging algorithm is proposed for visualization of very large scaled terrains. Using these algorithms a flexible terrain engine is designed and three dimensional landscape of Turkey is modeled. Some parts of this model is covered with detail textures, and texture organization is also supported with the proposed paging algorithm.

Quadtrees are one of the most popular hierarchical structures for continues level of detail rendering. Their level organization, during update process, depends on the camera position of the viewer. In order to give detail to rough areas, especially such as mountains, most of the current algorithms use a roughness criterion. This roughness criterion increases the detail of rough surfaces while it decreases the detail of smooth ones. Cumulating the triangulation of a terrain to different terrain portions may increase or decrease the frame rate according to the view angle of the camera. This is not a drawback for terrain implementations which are created for flight simulators, due to the restricted and slow movement of the camera. But in implementations where movement of camera is rapid and variable this is a real bottleneck. Even if we try to adaptively change the rendering resolution according to the camera angle we are unable to balance a constant frame rate. So discarding this roughness criterion is the best solution for real time applications.

In order to visualize the desired part of the terrain during navigation in real time, some paging algorithms have to be applied. The proposed paging algorithm in this thesis decreases the artifacts of background loading process, on the other hand due to its structure the quality of visible parts increases. In each altitude the detail distribution is smooth and does not disturb the viewer. The same paging organization is also used for detail texture mapping. Even the discrete time steps are dense the real time performance is not reduced. Even we use the proposed paging algorithm for scene map organization, on the borders of different detailed terrain patches some cracks may occur. Developed terrain engine does not erase background to take advantage of pixel mixing. Manually connecting the missing vertices within these cracks will be done as one of the future works. Another future work is to add lighting to the surface of terrain. Our current implementation uses global illumination, where

each vertex is illuminated equally. Using hardware assisted lighting techniques greatly reduces the real time performance due to the internal calculations, so manual calculated light values must be added to terrain. Fog and cloud effect is another area of interest which is essential for flight simulators and military applications.

The set of experiments show that proposed rendering and paging algorithm is quite applicable for real time applications. Rapid and free camera movement possibility causes the terrain engine adapted to each desired application and thus creates a flexible terrain structure.

REFERENCES

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

[2] **Lindstrom, P., and Pascucci, V.,** 2001. Visualization of Large Terrains Made Easy, *Proceedings of IEEE Visualization*, San Diego, California, October 2-26.

[3] **Lindstrom, P., Koller D., Hodges L. F., Ribarsky W., Faust N., and Turner,** 1995. Level-of-Detail Management for Real-Time Rendering of Phototextured Terrain, Technical Report GIT-GVU-95-06, January.

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

[5] **Reddy, M., Leclerc, Y., Iverson, L., and Bletter, N.,** 1999. Terra Vision II: Visualizing Massive Terrain Databases in VRML, *IEEE Computer Graphics & Applications*, March-April, pp. 30-38.

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

[7] **Möller, T., and Haines, E.,** 2002. Real-Time Rendering, A. K. Peters, Ltd., ISBN 1-56881-101-2.

[8] **Leclerc, Y., and Lau S. Q.,** 1994. Terra Vision: A Terrain Visualization System. Technical Report 540, SRI International, Menlo Park, CA., April.

[9] **Koller, D., Lindstrom, P., Ribarsky, W., Hodges, L. F., Faust, N., and Turner, G.,** 1996. Virtual GIS: A Real-Time 3D Geographic Information System, *Proceedings of Visualization '95*, IEEE Computer Society Press, pp. 94-100.

[10] **Neider, J., Davis, T. and Woo, M.,** 1993. OpenGL Programming Guide. Addison-Wesley.

[11] **Sorkine, O., Cohen, D., Goldenthal, R., and Lischinski D.,** 2002. Bounded-distortion Piecewise Mesh Parameterization, *IEEE Visualization*, July.

[12] **Larsen, B. D., Christensen, N. J.,** 2003. Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail, *Journal of WSCG*, vol. 11(2), pp. 282-289.

[13] **Pajarola, R. B.,** 1998. Access to Large Scale Terrain and Image Databases in Geoinformation Systems, Doctoral Thesis, Swiss Federal Institute of Technology (ETH) Zürich.

[14] **De Boer, W. H.,** 2000. Fast Terrain Rendering Using Geometrical Mipmapping. E-mersion Project, October.

[15] **Hoppe, H.,** 1996. Progressive Meshes, *Proceedings of SIGGRAPH 96*, ACM, pp. 99-108.

[16] **Snook, G.,** 2003. Real-Time 3D Terrain Engines using C++ and DirectX9, Charles River Media, Inc., ISBN 1-58450-204-5.

[17] **Polack, T.,** 2003. Focus on 3D Terrain Programming, Premier Press, ISBN: 1-59200-028-2.

[18] **Foley, J. D., Van Dam, A., Feiner, S. K., and Hughes, J. F.,** 1990. Computer Graphics: Principles and Practice, Addison-Wesley, ISBN 0-201-12110-7.

[19] **Aysal, T. C., Melkisetoğlu R., Celasun, I., and Tekalp, A. M.,** 2003. Design of Hierarchical 3D Meshes for Object Representation and Progressive Transmission, CCCT '03 and ISAS '03, Florida (USA), August

[20] **Pitas, I.,** 2000. Digital Image Processing Algorithms and Applications. John Wiley & Sons, Inc., ISBN: 0-471-37739-2.

[21] www.vterrain.org

APPENDIX A

Some screenshots from created terrain engine is given below.

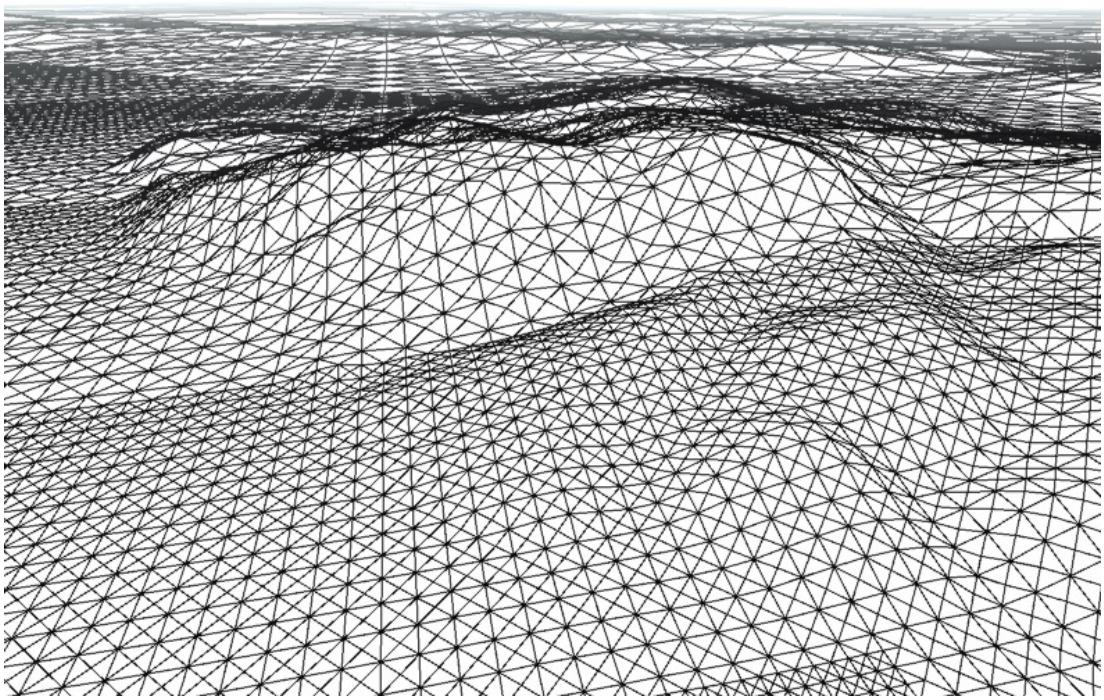


Fig. A.1 Wireframe view of terrain

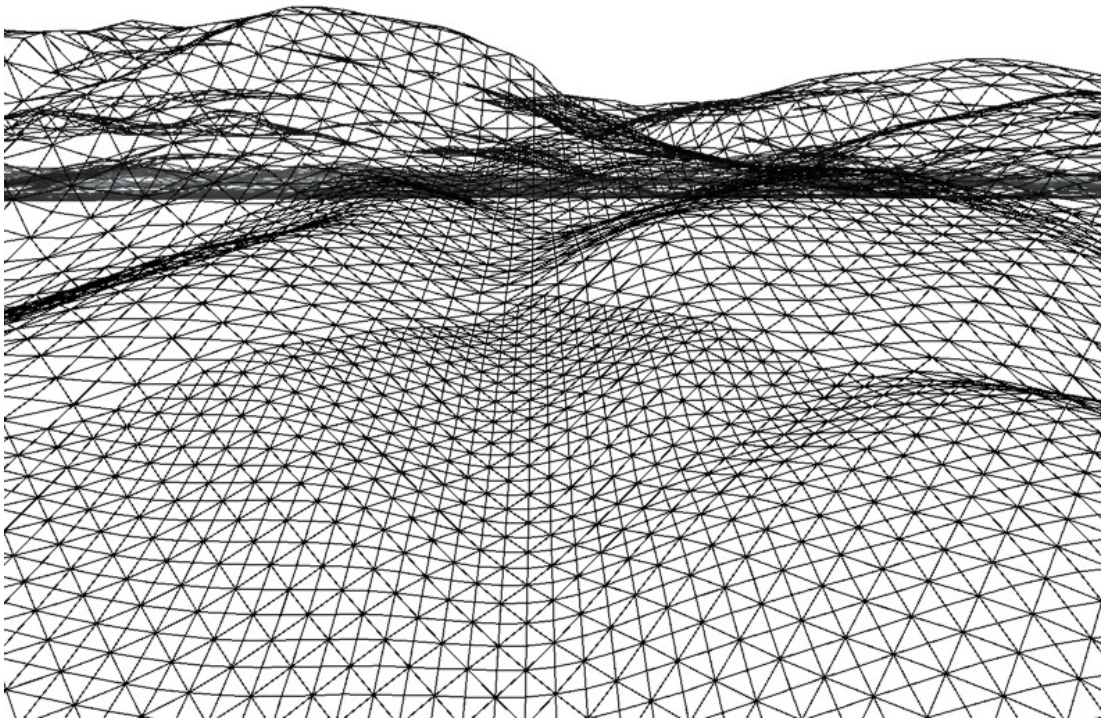


Fig. A.2 Wireframe view of terrain

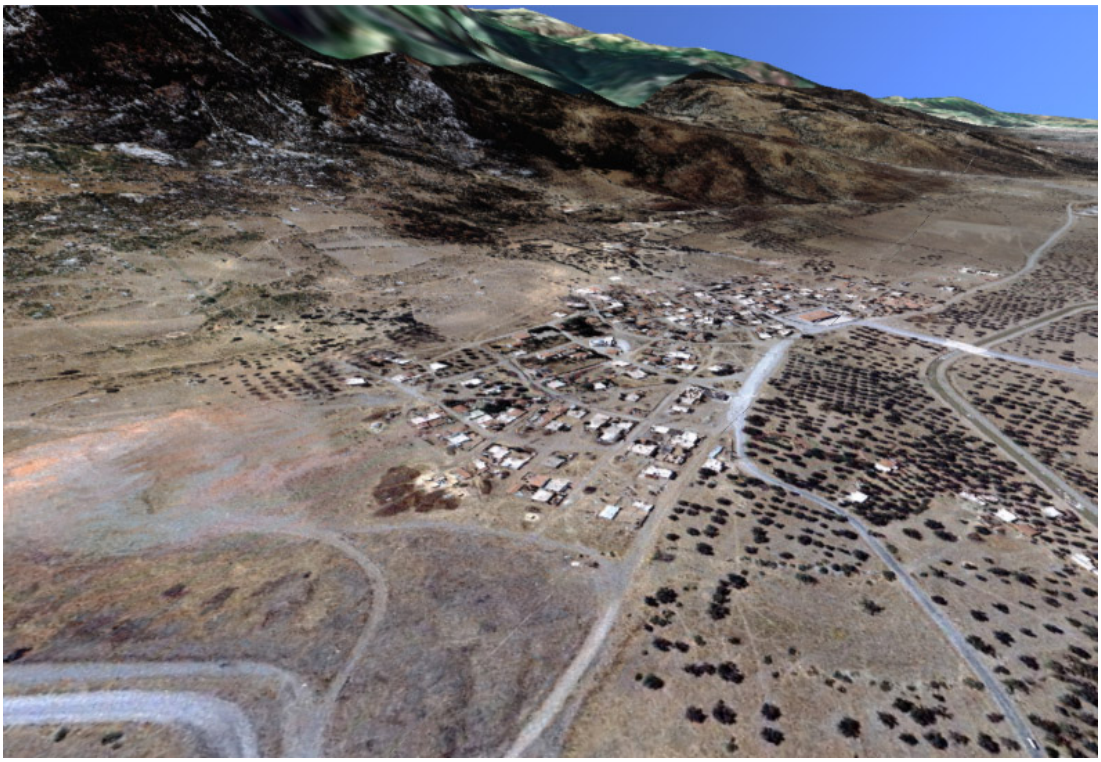


Fig. A.3 Detail textured terrain

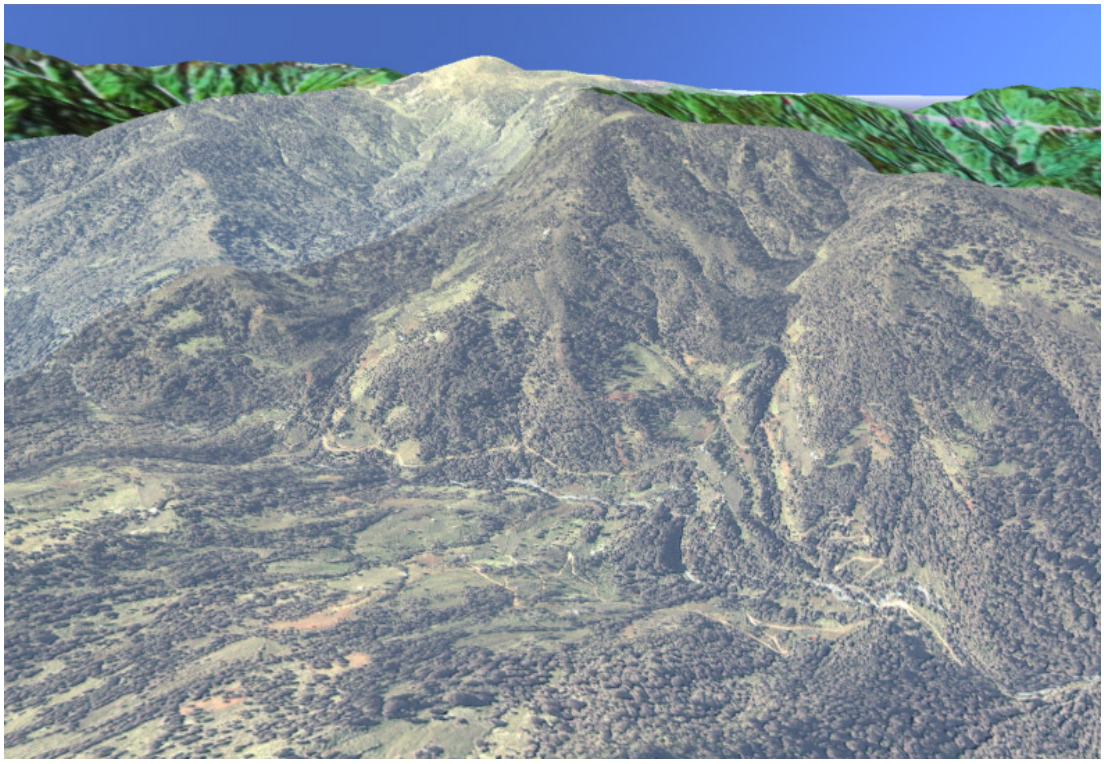


Fig. A.4 Detail textured terrain

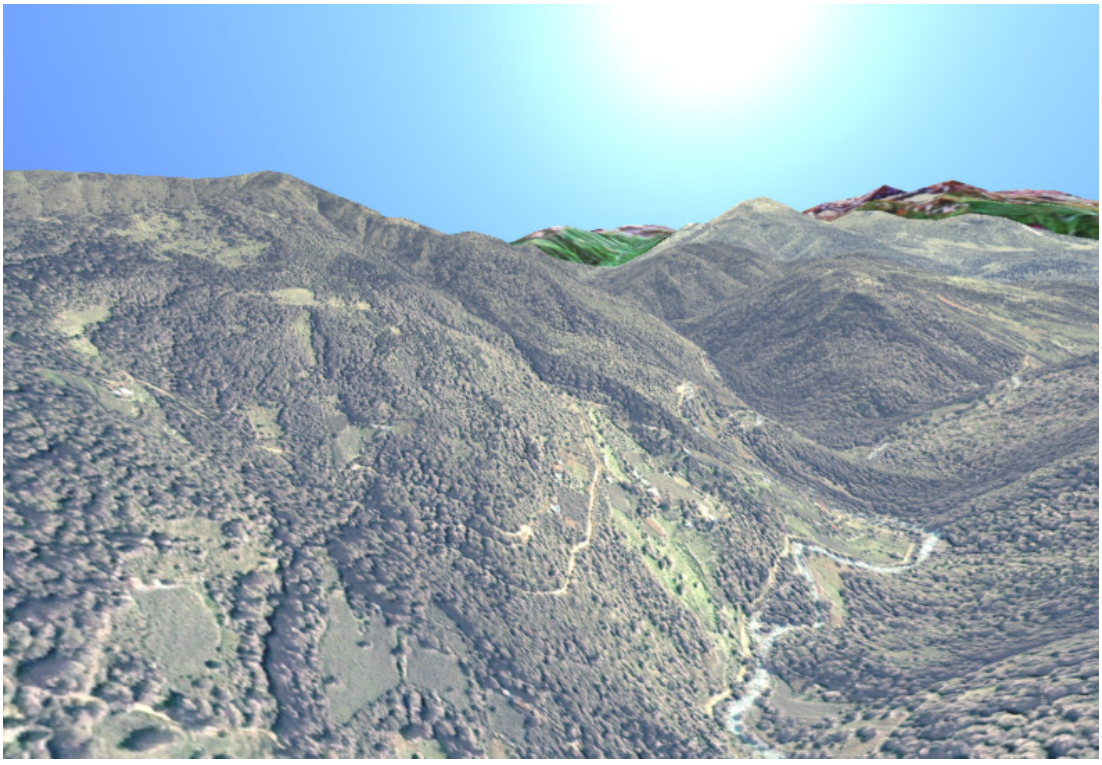


Fig. A.5 Detail textured terrain

AUTOBIOGRAPHY

Rupen MELKİSETOĞLU was born in Istanbul, Turkey in 1980. He graduated from Pangaltı Armenian Primary School and then he entered Getronagan Armenian High School and graduated from there in 1997. He continued his eduacation in Istanbul Technical University, Electronics and Communications Department and took his BSc degree in 2002 with honour. After graduating from university he was accepted to enter the Computer Science Department of Istanbul Technical University and from 2002 up to now, he has been working as a research assistant in Advanced Technologies in Engineering department in Istanbul Technical University.