INTERACTIVE EDITING OF COMPLEX TERRAINS ON PARALLEL GRAPHICS ARCHITECTURES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

UFUK GÜN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2009

Approval of the thesis:

INTERACTIVE EDITING OF COMPLEX TERRAINS ON PARALLEL GRAPHICS ARCHITECTURES

submitted by **UFUK GÜN** in partial fulfillment of the requirements for the degree of **Master of Sciences in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Müslüm Bozyiğit _____
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Veysi İşler _____
Supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Assoc. Prof. Dr. Halit Oğuztüzün _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Veysi İşler _____
Computer Engineering Dept., METU

Assist. Prof. Dr. Tolga Çapın _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Tolga Can _____
Computer Engineering Dept., METU

Mehmet Fatih Uluat (M.S.) _____
Software Engineer, Simsoft Bilgisayar Teknolojileri Ltd. Şti.

Date: 11.09.2009

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Ufuk Gün

Signature :

ABSTRACT

INTERACTIVE EDITING OF COMPLEX TERRAINS ON PARALLEL GRAPHICS ARCHITECTURES

Gün, Ufuk

M.S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Veysi İşler

September 2009, 88 pages

Rendering large terrains on large screens at interactive frame rates is a challenging area of computer graphics. In the last decade, real-time terrain rendering on large screens played a significant role in various simulations and virtual reality systems. To fulfill the demand of these systems, two software tools are developed. The first tool is a Terrain Editor that creates and manipulates large terrains. The second is a Multi-Display Viewer that displays the created terrains on multiple screens. Since the typical large terrains consist of many polygons, graphics boards might have difficulties in rendering the terrain at interactive frame rates. The common solution to this problem is to use terrain simplification without losing image quality. To this purpose, in this study, a paged level of detail mechanism that works with multiple threads is developed and integrated on multiple screen display systems to increase the performance of the high resolution systems.

Keywords: Parallel Rendering, Terrain Editing, Paged Level of Detail

ÖZ

KARMAŞIK ARAZİLERİN PARALEL GÖRSELLEME MİMARİSİYLE DÜZENLENMESİ

Gün, Ufuk
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Veysi İşler

Eylül 2009, 88 sayfa

Yüksek çözünürlüklü ekranlarda büyük arazileri yüksek çerçeve sayısında görselleştirmek bilgisayar grafiğinin zor konularından biridir. Son yıllarda, bu konu birçok simülasyon ve sanal gerçeklik sistemlerinin ihtiyacı haline gelmiştir. Bu talebi karşılamak için iki önemli uygulama geliştirilmiştir. Bunlardan ilki arazi yaratan ve şekillendiren Arazi Editörü, ikincisi ise bu arazileri çoklu ekranlarda görselleştiren Çoklu-Ekran Görselleştiricisidir. Böyle büyük araziler çok fazla çokgenden oluştuğu için, grafik kartları bu tür gerçek zamanlı çerçeve oranında görsellemekte sıkıntı yaşar. Bu sorunun genel çözümü arazinin görüntü kalitesinde kayıp yaşanmadan arazi üzerindeki dokuların ve üçgenlerin basitleştirilmesidir. Bu amaçla, bu çalışmada, arazileri parçalara ayıran ve bu parçaları farklı detay seviyelerinde görselleştiren bir mekanizma geliştirilmiştir. Daha sonra bu mekanizma çoklu ekranlı sistemle birleştirilerek yüksek çözünürlüklü hale getirilmiştir.

Anahtar Kelimeler: Paralel Görselleştirme, Arazi Editörü, Detay Çözünürlüğü

To My Family and My Fiancé

ACKNOWLEDGMENTS

The author wishes to gratefully thank his supervisor Assoc. Prof. Dr. Veysi İşler for his invaluable guidance, advice and encouragements for this research.

The technical assistance of Ph. D. student Mehmet Fatih Uluat is acknowledged.

This study is performed as a part of a professional project carried by a team of Simsoft Bilgisayar Teknolojileri Ltd. Şti. where the author was one of the team members. The author would like to thank management of Simsoft to provide the opportunity to take part in this project. The tools and techniques developed during this study are proprietary of Simsoft.

The author would also like to thank his managers at Simsoft, Mrs. Gökçe Yıldırım, Mr. İsmail Bıkmaz, Mr. Selman Duatepe and Mr. Cemal Koplay for their comments and all kinds of support.

The author would like to thank TÜBİTAK for its financial support.

Finally, the author wishes to express his special thanks to his family and his fiancée for their patience, support and motivation.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

Abbreviation or Symbol	Text
ALU	Arithmetic and Logic Unit
CG	Computer Graphics
CPU	Central Processing Unit
GIS	Geographical Information System
GPU	Graphics Processing Unit
IP	Internet Protocol
LOD	Level of Detail
OpenGL	Open Graphics Library
OSG	Open Scene Graph
PC	Personal Computer
TCP	Transmission Control Protocol
VBO	Vertex Buffer Object

CHAPTER 1

INTRODUCTION

In the last decade, real-time terrain rendering has played a significant role in various areas such as Geographical Information Systems (GIS), military, flight and driving simulations, computer games, computer animation and virtual reality. Achieving high frame rates is essential for these kinds of applications to be effective. Rendering large terrains on large screens at interactive frame rates is a challenging area of computer graphics. Both creating large terrains and rendering these terrains on large screens are individually demanding processes. These large terrains might be built using a real world data or a generic height map. A generic height map can be created by either using a terrain generation algorithm or a tool which provides user instruments to manipulate terrains.

A terrain editor should include features like height field editing, texture painting and object placing, such as roads, rivers, lakes or vegetation. Generally height fields are manipulated with brushes, for instances raise, lower, smooth or flatten. Calculating the new height values and rendering the results at interactive frame rates is a challenging issue. To this purpose, the Vertex Buffer Object (VBO) is used in order to render and edit terrain which increases the frame rates. VBO is a feature that allows the tool to store the vertex data in application memory [22]. Another important feature that is expected from a terrain editor is texture painting which is implemented by the mixture of alpha maps, tile, base and slope textures. Alpha maps display the placement of tile textures and the base texture is used in order to decrease the repeating effect of this tiling. Base texture is also used to roughly describe the shape of terrain. Slope textures are used in regions where the slope is close to 1. Some other texturing and coloring mechanisms are also developed and mentioned in Chapter 2.

Sometimes the size of the terrain data might be larger than the main memory can store. One possible solution to this case is paging the terrain where the entire terrain data is divided into pages and the pages are loaded into memory as they are needed. This approach is employed in this study. Blocks are paged in full resolution while they are in editing window which represents the enabled editing area, but the other blocks which lay outside the editing window are not rendered. Loading and unloading new pages is triggered by the moving of this editing window.

In this study, a multi-screen terrain viewer which displays the results of the terrain editor is also developed. Multiple screens can be achieved by the organization of different computers or different graphics boards on a single computer. The number of screens, CPUs and GPUs are configured prior to start of application. In this project, increase in display resolution is achieved with the help of multiple screens. This feature can also be used in cubic display systems like domes or caves to increase the realism. A dome is defined in dictionaries as a structural element of architecture that resembles the hollow upper half of a sphere. In dome systems, a display system projects the scene to the walls of this dome. A person who stands in the center of this dome feels as if he is standing in a real world. Similarly, a cave system which is a cube with 6 surfaces gives a realistic view as domes.

During the development of the terrain viewer, a paged level of detail mechanism is designed and implemented. Terrain tiles are paged in and out in threads that calculate the new resolution of terrains and create them in different levels of detail. The level of detail of pages is applied not only on mesh geometry, but also on textures of terrains. The significant impact of performance can be seen in the Chapter 4.

The following parts contain chapter that give more detailed information about this study. The relevant literature of the terrain editor is discussed in Chapter 2. Chapter 3 introduces the algorithms of terrain rendering. Parallel rendering

algorithms and APIs are discussed in Chapter 4 and the thesis continues with Chapter 5 which explains how the terrain editor and the multi-screen viewer are developed. The summary and the suggestions for further works are presented in Chapter 6.

CHAPTER 2

A SURVEY OF TERRAIN EDITORS

There are numerous COTS (Commercial, Off the Shelf) terrain editors which manipulate terrain geometry, paint with textures and place objects on terrains. Each editor uses different techniques to render terrain geometry. Below, several terrain editors and their features are discussed and compared with respect to each other.

2.1 FreeWorld3D

FreeWorld3D is an interactive 3D terrain editor that uses OpenGL. It is mostly designed for beginners or independent game developers. It can produce whole 3d terrain with the environment on it [15]. Screen shots of the application can be seen from Figure 1 and Figure 2. Features of FreeWorld3D can be summarized as below:

- Mesh Editing
    - Massive Terrain Creation
    - Raise, Lower, Smooth, Flatten Tools
    - Circle and Square Tool Shapes
    - Automatic Terrain Generation
- Terrain Painting
    - Paint terrain with textures
    - Paint terrain by slope
    - Paint terrain by height
- Object Placement
- Road Editing
- Vegetation Painting

- Water Editing

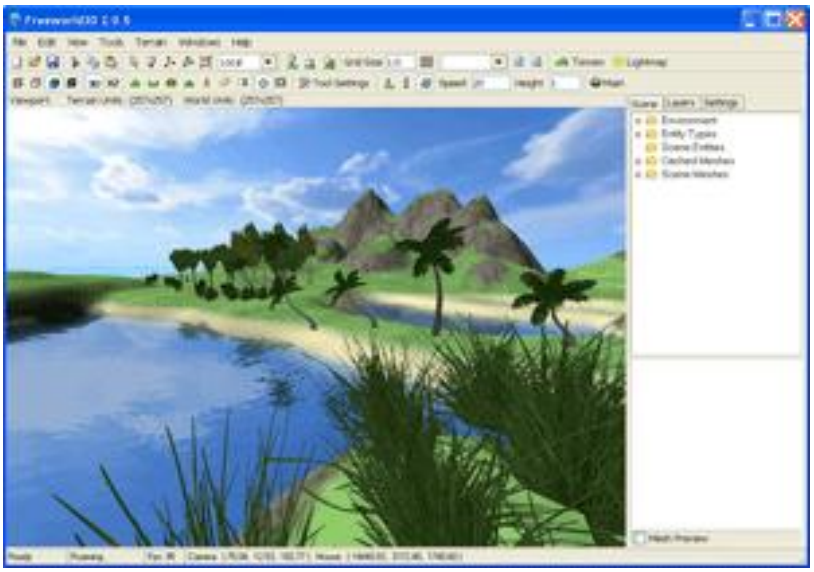

Figure 1 Screenshot of FreeWorld 3D. Final view of the terrain can be seen on the left side of the figure.

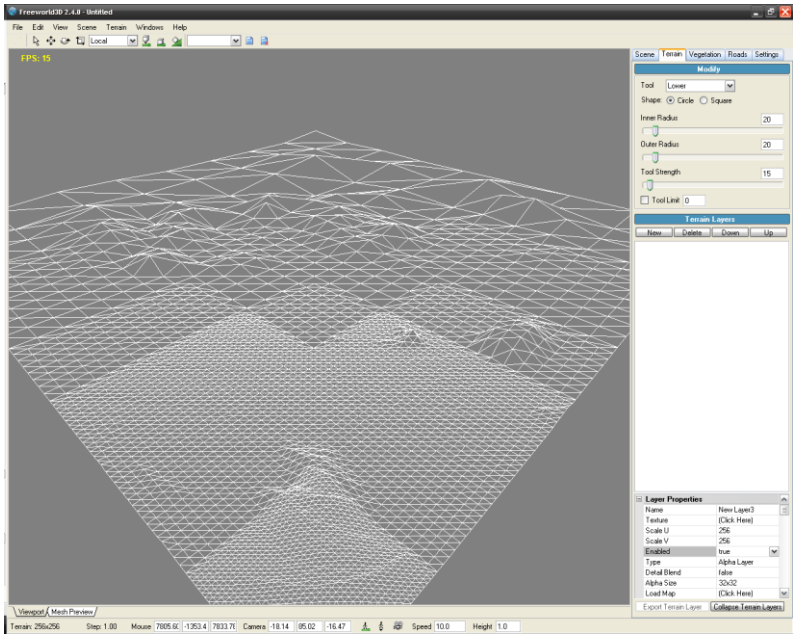

Figure 2 Screenshot of FreeWorld 3D in wireframe mode. Closer regions on the terrain are represented with more triangles than the farther regions of the terrain.

FreeWorld3D uses “Tiled Blocks” algorithm [Section 3.4] to render the terrain.

As you can see from Figure 2, closer parts of the terrain are rendered fine and farther parts of the terrain rendered coarse.

2.2 EarthSculptor

EarthSculptor is a real-time terrain editor and texturing tool for rapid development of 3D landscapes, multimedia and game development. EarthSculptor is developed in OpenGL for Windows, using a T-Strip LOD algorithm. T-Strip LOD algorithm renders large terrains efficiently and it enables geo-morphing, decals, fast collision detection and lighting [18] [17]. Screen shot from EarthSculptor are shown in Figure 3 and Figure 4. Its features can be summarized as below:

- Enables terrain tiles.
- Terrain Editing
    - Raise, lower, level, grab, smooth, erode, push and ramp
- Terrain Texturing
- Plug-in system

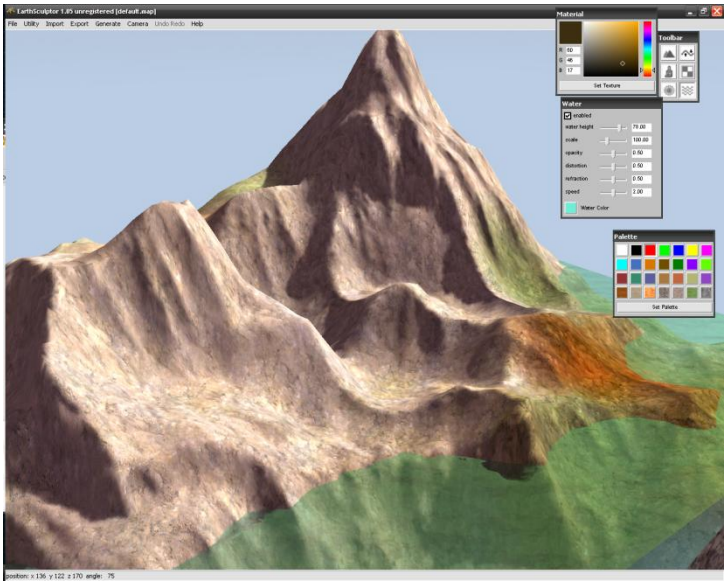

Figure 3 Screenshot of Earth Sculpture in solid view. Created terrain can be seen on the larger window. Editor widgets are placed on the right side of the window.

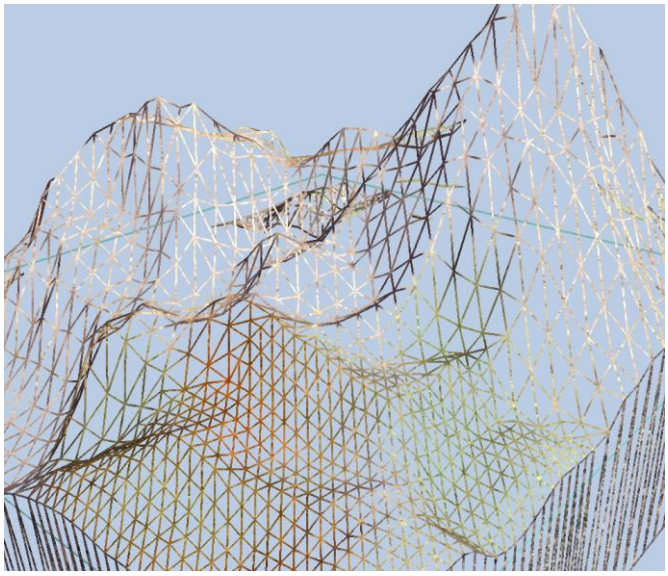

Figure 4 Screenshot of Earth Sculpture in wireframe view. Closer regions are rendered with more triangles and farther regions are rendered with fewer triangles.

The most important advantage of T-Strip LOD algorithm which is used in this tool is that there is no popping during walkthrough. In the other LOD techniques, when the camera moves, vertices are changing by the sampled height field. That makes popping effects on level changes.

In T-Strip LOD, vertices are being changed by the interpolated height field. This method gives an impression that the terrain level of detail changes very smooth but the process needed for this algorithm at behind is too much. The interpolated height of all vertices on the scene is calculated on every frame.

This algorithm provides very smooth transitions on the visualization of different level of details when there is only one page. When there is more than one page, the cracks occur and the management of the algorithm becomes very difficult.

2.3 PnP TerrainCreator

PnP TerrainCreator is a real-time interactive terrain editor. This tool enables users to change height maps with some filters, paint surface textures and paint vegetation on terrains.

The rendering algorithm of this tool is not good enough for large terrains. There is no LOD method or optimization in PnP TerrainCreator. It uses pure OpenGL in order to render scene.

The most important advantage of this tool is plug-in architecture. Users can easily add their prebuilt plug-in to the application and use it on the system. This plug-in can be an object editor, vegetation painter or something else. This property makes this tool very adaptable for future needs. A screen shot of the application is shown in Figure 5. The features of PnP TerrainCreator are as follows: [19]

- Interactive Heightmap Editing
    - Randomizing, smoothing, raise, lower, flatten
    - It also enables users to integrate their own manipulation brushes.
    - Both 2D and 3D editing
- Surface Texturing
- Seas and Oceans
- Static Objects
- Terrain Vegetation
- Environmental Sounds

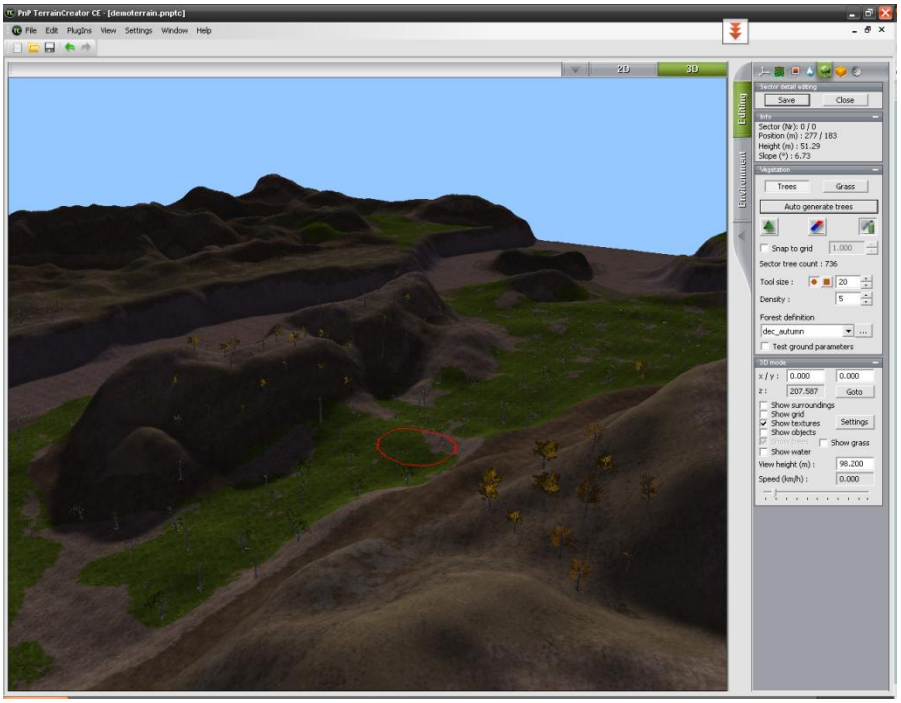

Figure 5 A screenshot of PnP Terrain Creator. The final view of the terrain is shown on the left side and the editing brushes are shown on the right side.

2.4 Artifex Terra 3D

Artifex Terra 3D is a Heightmap based real-time terrain editor and texture painter.[20] This terrain editor is an Ogre based application where Ogre is an open source graphics engine. A screen shot of the application is shown in Figure 6. Artifex Terra 3D uses Bin-tree Hierarchies LOD technique in order to render terrain which can be seen from Figure **7**. It uses fewer triangles for plane parts and more triangles for the detailed parts. Its features can be summarized as below:

- Terrain editing
    - Lower, Raise, Blur, Flat, Blur, Boil, Plane.
- Texture painting with some tools and filters
    - Paint, Erase, Darken, Brighten, Blur, Sharpen, Contrast, Noise
    - Gaussian Blur, Dilate, Edge, Erode, Repair, Unsharp Mask, Sharpen, Noise, Clear
- Paged and animated grass support.

- Object placement
- Environment settings
- Water Editor

Figure 6 A screenshot of Artifex Terra 3D

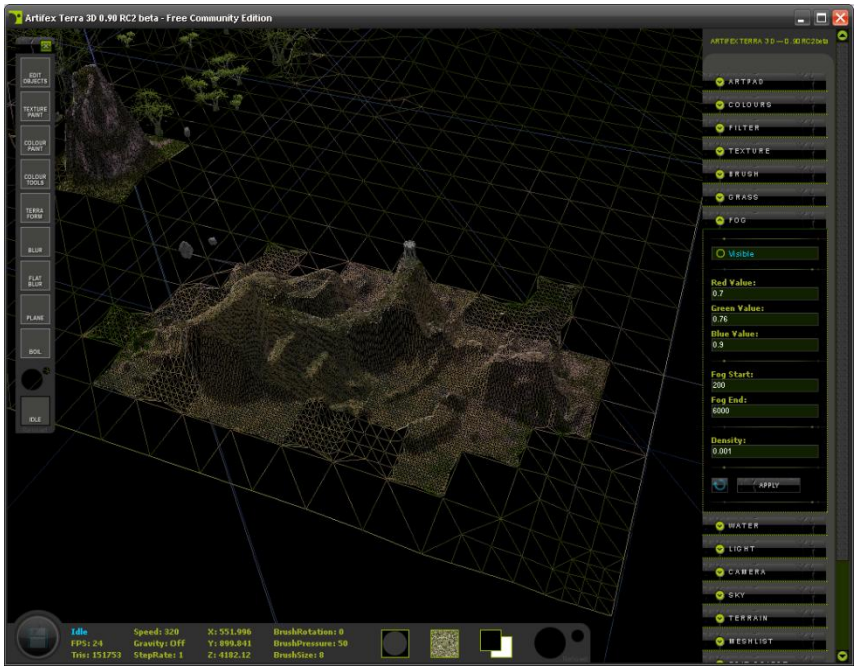

Figure 7 **A screenshot of Artifex Terra 3D in wireframe mode. High detailed regions are represented with more triangles.**

2.5 Creator Terrain Studio

Creator Terrain Studio is generated by a company named MultiGen-Paradigm. It can create complex terrains and environments as in Figure 8. A screen shot of the application on run time is shown in Figure 9. The features of Creator Terrain Studio can be summarized as below:

- Terrain editing
- Texture painting with some tools and filters
- Object placer
- Environment settings
- GIS integration

Figure 8 A large terrain created with Creator Terrain Studio.

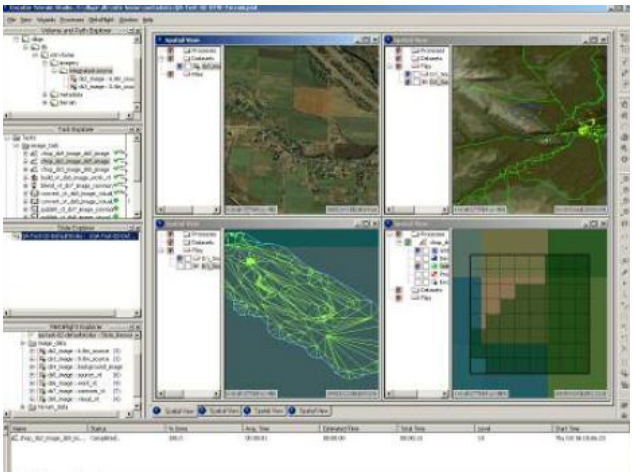

Figure 9 **A screen shot of the tool with multiple views.**

2.6 Comparison

A feature table of editors is given in the Table 1. Terrain editors are given in the first column and the capabilities of the features on the right columns. Mostly terrain editors render terrains with LOD. Paging mechanism can be seen in some editors, but GIS integration and large-area rendering is very rare. Parallel rendering is never used before in any terrain editors.

Terrain Editor	LOD	Paging	GIS Integration	Parallel Rendering	Large-area Terrain
FreeWorld3D	yes	no	no	no	no
EarthSculptor	yes	no	no	no	
PnP TerrainCreator	yes	yes	no	no	no
Artifex Terra 3D	yes	no	no	no	no
Creator Terrain Studio	yes	yes	yes	no	yes

Table 1 The feature table of editors

Please notice that the most of these editors have limited resolutions in order to edit terrain. These editors have mostly focused on small terrains. Since this study focuses on large terrains, the algorithms they used will not work to our case. To this purpose, new rendering and LOD mechanisms should be developed to improve the performance of rendering large terrains.

The current editors mostly do not have GIS integration. Nowadays, since working on real data is demanded by simulation systems, importance of GIS systems is increased. GIS integration is planned for the application which is developed in this study.

There is no parallel rendering architecture in the current editors which can be seen from the Table 1. That means these editors have a limited screen resolution on rendering terrain or limited terrain complexity. In this study, a parallel rendering mechanism has been developed to render terrain on multiple screens to increase the display resolutions.

CHAPTER 3

TERRAIN RENDERING

A terrain is usually represented as a height map as we assumed in this thesis. A height map which is sampled from terrain data should be converted to polygons prior to being fed into graphics hardware. Mostly, this process is composed of the triangulation of every three points on the height data. In large terrains, there are so many triangles that a hardware cannot handle directly in order to achieve enough frame rates to render. Increasing resolution causes an observable decrease on performance.

The common approach in order to increase the rendering performance is decreasing the number of polygons on terrain surface. In the following sections, we present a survey of the terrain rendering methods which aim the minimization of number of polygons without losing image quality.

3.1 Irregular Meshes

Irregular meshes method is also known as triangulated irregular networks which provide the best approximation for the triangulation of terrain surface. This method represents the surface by triangles which are different size and shapes [10]. In Figure 10, it is seen that high detailed surfaces are represented with many small triangles and relatively low detailed surfaces are represented with large triangles.

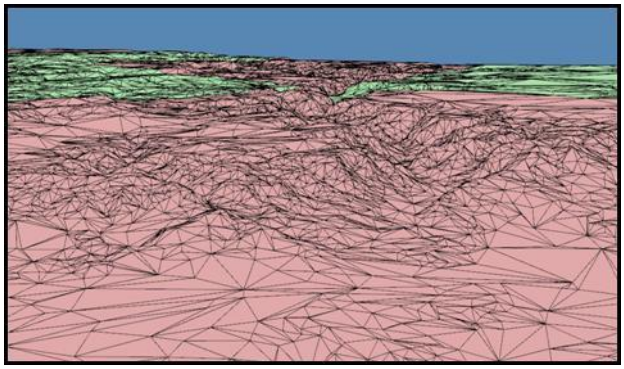

Figure 10 An example output of irregular mesh algorithm. High detailed regions are represented with more triangles than low detailed regions.

This technique uses too much CPU and memory to model the new mesh [10]. Moreover, irregular structure makes geometry caching difficult. Since modern rendering systems are so integrated with GPU, this algorithm of terrain-rendering is not suitable anymore [10].

3.2 Bin-tree Hierarchies

This hierarchical terrain-rendering algorithm recursively subdivides the height map into right triangles resulting in a hierarchy. Each triangle can be split into two right-isosceles triangles of half the size, by dividing along a line from the apex to the midpoint of the hypotenuse. The inverse of this process is called *merging*. This binary tree stored in memory. These triangles in the tree are split or merged by their need in the priority queue. In Figure 11, many triangles which are divided into two pieces can be seen easily. In highly detailed regions of the terrain, these triangles are divided much more recursively relative to the flat regions of the terrain.

One advantage of this method is that it allows adaptive refinement. Terrain regions can be represented with more or fewer triangles when needed and this can be done so fast as the viewpoint moves. This method has also very accurate approximations of the terrain with fewer triangles. Additionally, it is very easy to

traverse. It is also beneficial for view-frustum and occlusion culling. If a parent triangle is out of the frustum then all children will not be visible.

On the other hand, hierarchical techniques have some disadvantages. The calculation of the detail requires CPU computation and significant amount of memory to track the current state of triangulation. Reducing the number of triangles is not the best approach on modern graphics boards. It should be aimed to maximize the number of triangles that send to GPU in order to render terrain. Since the triangulation is not uniform, it is difficult to form triangle strips. An example output of Bin-tree Hierarchies algorithm is shown in Figure 11.

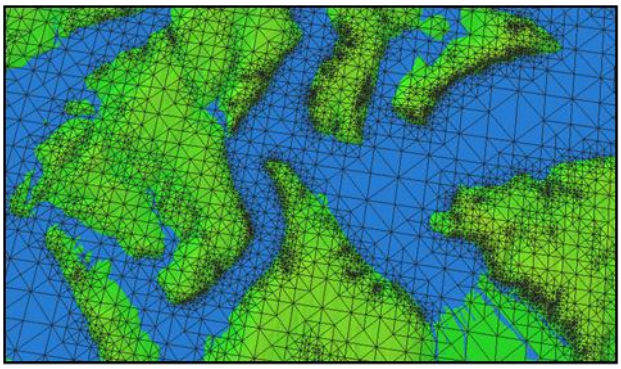

Figure 11 An example output of Bin-tree Hierarchies Algorithm. Smaller triangles represent more detailed regions.

3.3 Bin-tree Regions

Bin-tree regions are based on refinement operations on large regions associated with a bin-tree structure.[8] After the triangulation of precomputed regions, they are uploaded to the buffers of video memory [9] [7]. In order to avoid cracks, some approximation techniques are applied to smooth the terrain.

An example output of Bin-tree Regions is shown in Figure 12.

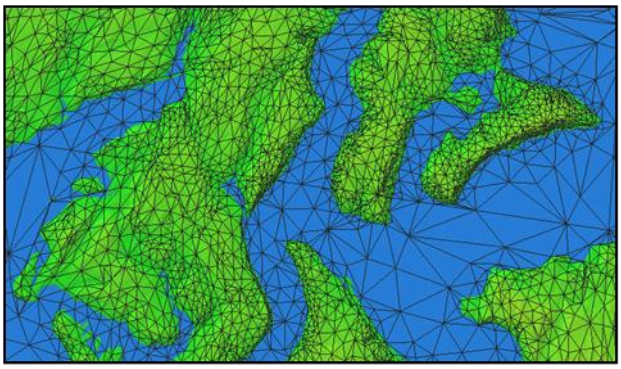

Figure 12 An example output of Bin-tree Regions Algorithm. High detailed regions are represented with more triangles than low detailed regions.

3.4 Tiled Blocks

Rendering terrain with tiled blocks aims to render square patches. Each square patch has different resolution. The resolution of the surfaces decreases by the distance to the viewer increases. Terrain data is stored in a set of uniform 2D grids. Around the viewer, there are nested grids. Grid resolutions differ by a factor of two at each refinement transition. The final structure of the terrain can be Figure 13. The closer regions are rendered at higher resolution with respect to farther regions.

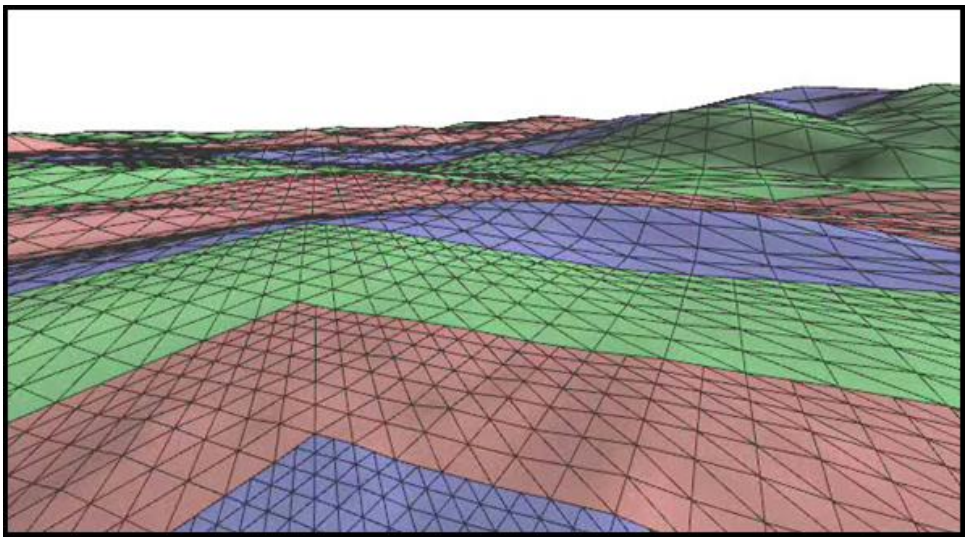

Figure 13 An example output of Tiled Blocks algorithm [7]. Closer regions are represented with more triangles and farther regions are represented with fewer triangles.

The grid structure has many advantages comparing to irregular mesh techniques. The most important advantage is that the data structure of the method is very simple. It also provides a steady rendering rate, even if viewpoint moves. Visual transitions are smoother and the compression is much more efficient [10].

This method focuses on performing as much operations as on the GPU than on the CPU. Since the modern GPUs can calculate most of the operations better than CPUs, it is suitable for GPU rendering.

The main challenge in this approach is rendering block boundaries seamlessly among blocks. Since every block has different resolution, there is a probability that some cracks occur along the block borders [7].

3.5 Triangle Strip preserving LOD (T-Strip LOD)

T-Strip LOD is a method that preserves the height map as a series of triangle strips [17]. Generally the other LOD techniques subdivide the triangles of mesh to get more detail in the closest mesh to the viewer and less detail for farther away. (See Figure 14) In T-Strip, instead of using the subdivision of triangles, it is using the subdivision of lines. Retaining the geometric structure of terrain makes it faster to draw the triangles as a triangle strip. A triangle strip is a geometric structure that renders triangles whose vertices follow a path which make a series of triangles [16].

This algorithm recursively splits a line segment into 2 equally sized smaller line segments. It starts from the largest blocks size which is entire terrain and it goes down to the smallest block size. Every split reduces the block size by half. It is similar to the algorithm in quad trees but there is a difference. In quad tree splitting is done on 4 quads and here splitting is done one a horizontal line segment. This recursive step is applied until LOD has been satisfied. Individual lines make a continuing path of triangles which form a triangle strip adding an extra performance on rendering [17].

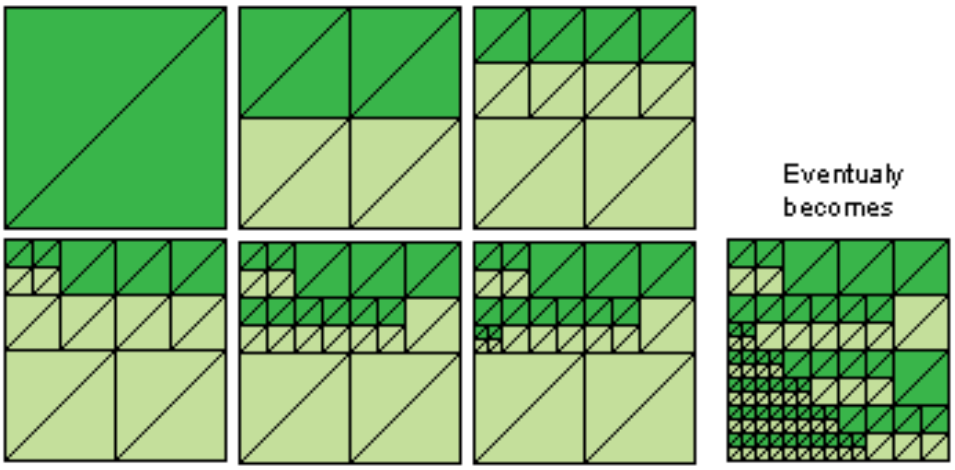

Figure 14 The subdivision of Lines (top) as Triangle Strips (bottom) [17]

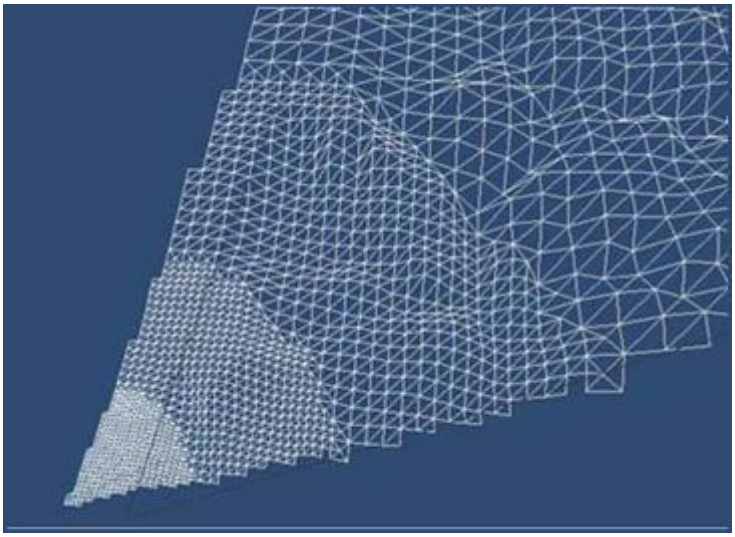

Figure 15 An example view of T-Strip LOD Algorithm [17]. The number of triangles decreases since the distance to camera increases.

For each line segment, the algorithm decides whether the line will fully split, not split or partially split but the split remains contiguous [17].

CHAPTER 4

PARALLEL RENDERING

Parallel rendering is a method used to improve performance of the CG (Computer Graphics) software. When high-quality images with high frame rates are required or when the scene is complex, importance of rendering process increases. To achieve necessary level of performance, parallel computing techniques are used.

There are several approaches for parallel rendering, namely “sort-first”, “sort-middle” and “sort-last” by considering the distribution time of data.

4.1 Sort-First

Methods based on Sort-First rendering divides the screen into tiles and every tile is rendered on a different node [11]. In this algorithm, a full graphics pipeline assigned for each tile. Since sort-first methods provide linear speed up on rendering performance, it is the best way of increasing the image resolution.

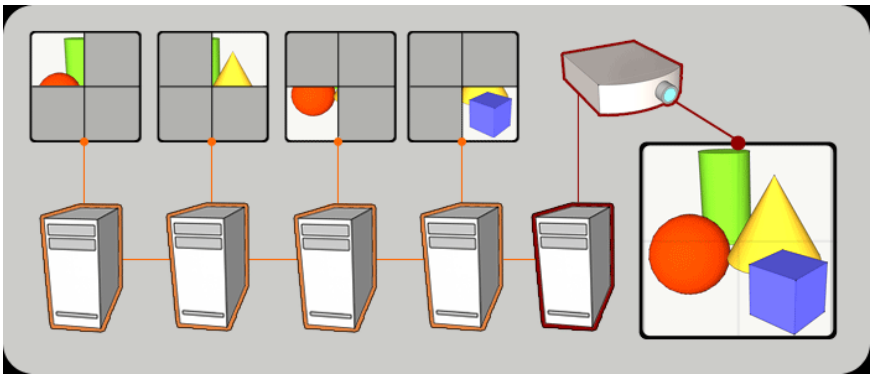

Figure 16 A sample structure of sort-first [12]. The viewports of the scene are calculated on render nodes and merged on control node.

An example of sort-first method is shown in Figure 16. It is also known as image decomposition mode. Each rendering machine calculates the pixels of a part that it is in charge and the final machine composes the outputs of each rendering machines [12].

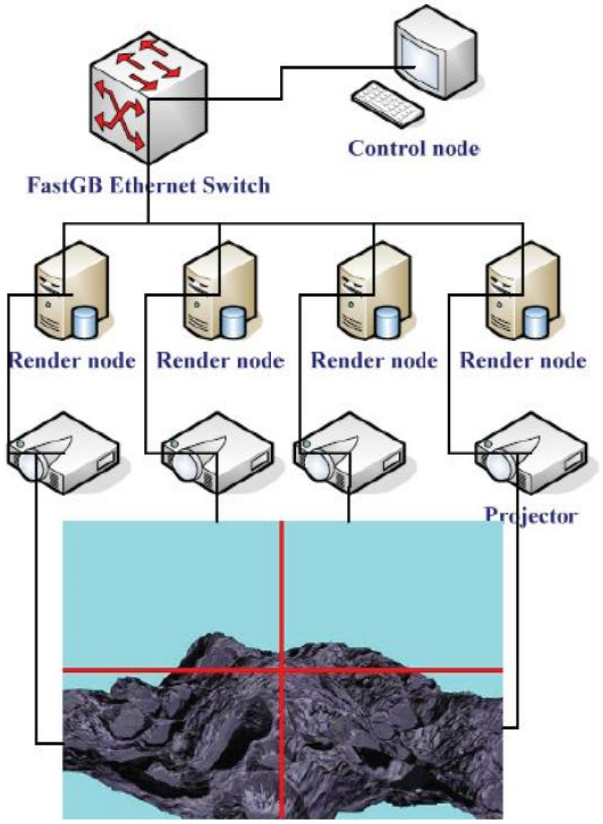

Figure 17 Another sample structure of sort-first [11]. Control node synchronizes the output of render nodes.

Sort-First methods can also be applied as in Figure 17. Every node renders different part of the final view in the screen space and displays it on the monitor. As you can see, in sort-first method rendering nodes does not only send final image to controller machine in order to render scene as in Figure 16 but also display partial scene on its own monitors as in Figure 17.

4.2 Sort-Last

Sort-Last is also called as database decomposition method [12]. It decomposes the scene elements across all rendering units and merges partially rendered frames on final view [11] (See Figure 18). Every unit has its own private frame buffer and these frame buffers recompose on viewer machine. Sort-Last method is well suited for the applications whose scene data contains a large number of polygons [12].

Even if this method scales the rendering very well, the recomposition process is very expensive since there is a large amount of pixels. For each pixel, there should be a calculation for depth analysis of objects.

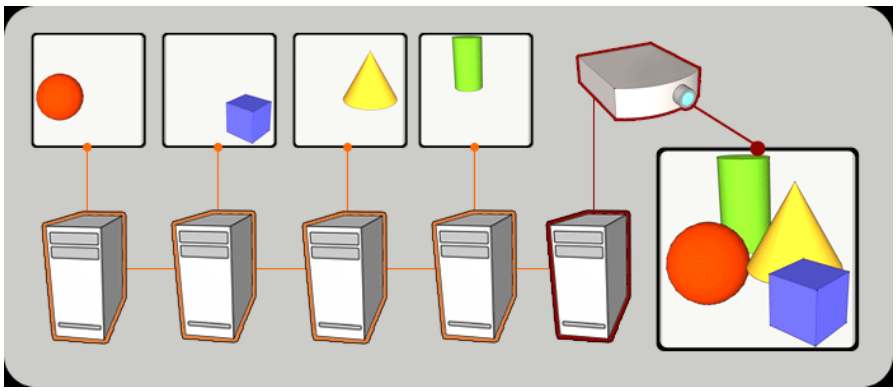

Figure 18 A sample structure of sort-last [12]. Render nodes which are placed on the left side of figure calculates different objects and control node which is on the right merges the results.

4.3 Sort-Middle

Sort-Middle rendering distributes triangles among geometry units arbitrarily, and then sends the transformed triangles to the renderers depending on a screen position [11]. It is a hybrid model of Sort-First and Sort-Last methods.

4.4 The comparison of sorting methods

Sort-First methods are mainly used in order to increase the screen resolution or displaying larger areas of the scene at the same time. Sort-Last methods are chosen when there is a performance demand on a single monitor. Since the objective of this study is displaying scene on multiple monitors, methods of sort-first approach are used.

In the following sections, some detail is given on major parallel rendering APIs.

4.5 Parallel Rendering APIs

4.5.1 WireGL

It is the first sort-first parallel rendering system for a PC cluster which is developed in Stanford University [21]. WireGL uses standard client/server model. Clients provide graphics source and servers get the commands from network and render them. WireGL helps graphics developers to make applications in parallel mode. Figure **19** is the diagram that shows how WireGL system works.

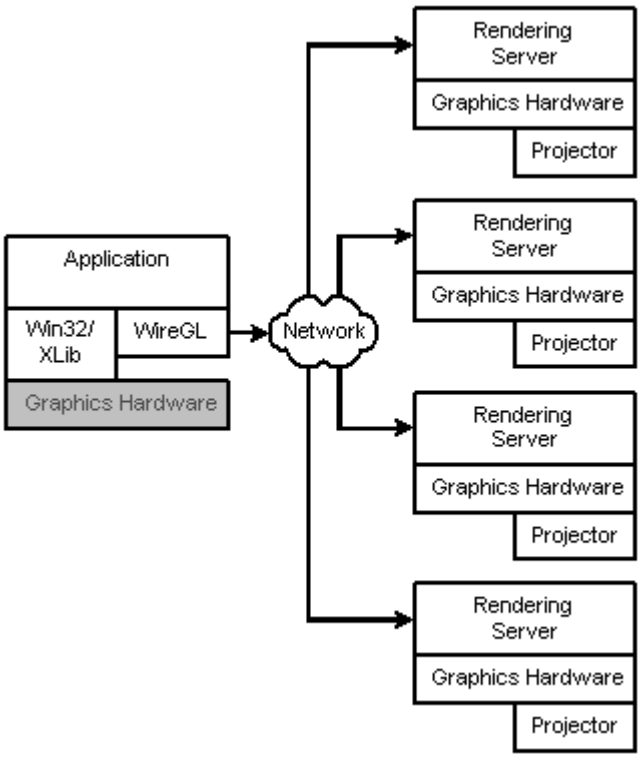

Figure 19 **The diagram of system architecture [87]**

4.5.2 Chromium

Chromium is derived from the project WireGL and it is also developed by Stanford University. Chromium is an interactive rendering system on the clusters of workstations. Various parallel rendering algorithms like sort-first and sort-last can be implemented with Chromium. It also allows filtering and the manipulation of OpenGL command streams for non-invasive rendering algorithms [4] [5].

4.5.3 OpenGL Multipipe SDK (MPK)

OpenGL Multipipe SDK is an API for OpenGL that is designed to help developers meet the demand of immersive environments [13]. This product makes the application scalable by additional pipes and other scalable graphics hardware [14].

OpenGL Multipipe SDK provides these features [13]:

- Run-time configurability: The application can be run in one window on a single workstation or run on multiple pipes on a scalable machine.
- Run-time scalability: Rendering can be done on distributed machines.
- Scalable graphics hardware support is integrated.
- Stereo and Immersive Environments support is integrated.

4.5.4 Equalizer

Equalizer is a toolkit for scalable parallel rendering based on OpenGL which provides an API to develop scalable graphics applications for wide range of systems [1]. It enables applications to use multiple graphics boards, processors or computers to scale rendering performace, visual quality and display size. Chromium and Equalizer solve a similar problem by enabling applications to use multiple GPU's but their characteristics and use cases are quite different.

Equalizer is much more scalable, flexible and compatible with less implementation overhead [1][2].

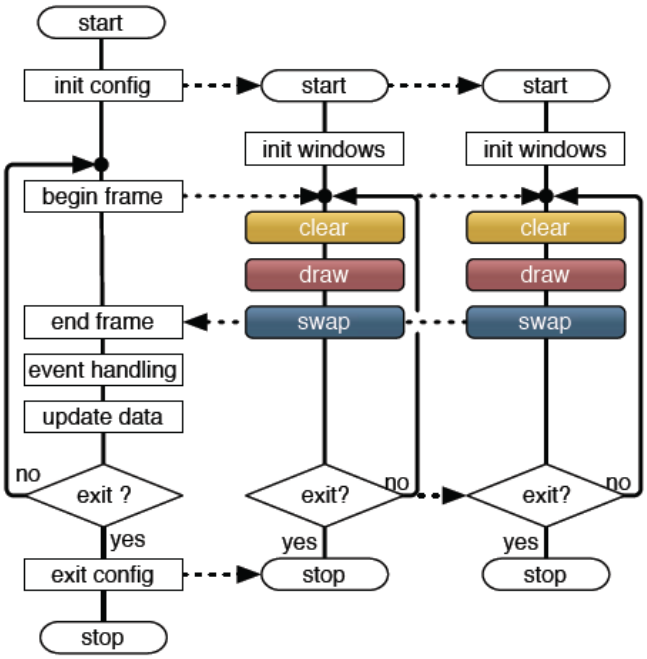

Figure 20 The illustration of the basic principle of any parallel rendering application [86]

Figure 20 illustrates the basic principle of any parallel rendering application. The typical OpenGL based application has an event loop which redraws the scene, updates data in order of received events, and then redraws the new frame. A parallel rendering application uses the same model but it seperates the rendering

code from main event loop. The rendering code is executed in parallel on different resources. Equalizer follows this model which makes the application development as easy as possible [6].

Since Equalizar is found to be more capable and advantougous with respect to the other APIs, we give much more detail on equalizer by dercribing its interface. At the following parts the system architecture is presented in headings “Interface”, “Application” “Rendering Client” and “Equalizer Server”.

4.5.4.1 Interface

Equalizer is a parallel rendering framework using a similar underlying concept as OpenGL Multipipe SDK (MPK) but there are some architectural improvements compared to MPK.

Equalizer provides a framework to facilitate the development of distributed and non-distributed parallel rendering applications. Programming interface is based on a set of C++ classes, modeled closely to the hierarchical resource description used by the server. The application subclasses these objects and overrides C++ task method, similar to C callbacks. Framework calls these methods in parallel, depending on a current configuration [4].

Unlike MPK, an Equalizer application does not select the rendering configuration itself. System-wide configuration server selects rendering clients by the guideline of user. Rendering clients provided by the application are launched and controlled by the server.

On a higher level, Equalizer uses a client-server approach built on a peer-to-peer network layer. It uses TCP/IP sockets in order to message or transmit the result images.

4.5.4.2 Application

The application in Equalizer does not execute any rendering but triggers the rendering loop only. The application process may host one or more rendering clients. If a configuration has no additional nodes besides the application node, all application code is executed in the same process without any network data distribution [4].

The application provides a rendering client during the initialization of the server. The rendering client is mostly the same executable as the application. This client is deployed on all nodes specified in the configuration by the server. The main rendering loop is simple:

- The application requests a new frame to be rendered.
- Synchronizes on the completion of a frame
- Process events from rendering clients.

Figure 21 shows the simplified execution model of an Equalizer application.

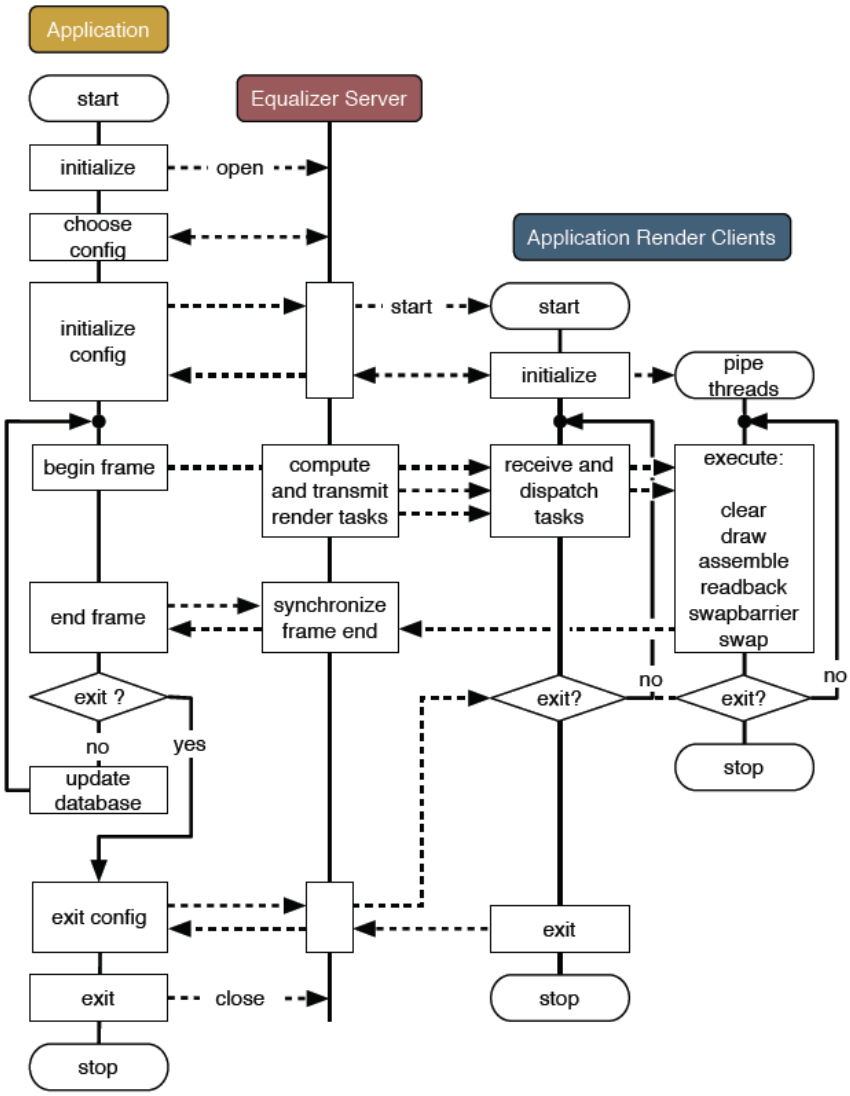

Figure 21 The simplified execution flow of an equalizer application [86]

4.5.4.3 Rendering Client

Every Equalizer application has a rendering client which can be the same executable as the application itself. Since rendering client does not have main loop, it is controlled by the Equalizer framework. A rendering client consists of the following threads:

- 1 x main thread
- 1 x network receive thread
- 1 x pipe thread for each pipe to execute rendering tasks.

On client library the main loop is implemented, which receives network events and process them. The network data contains rendering tasks parameters computed by the server. Rendering client sets up the scene by these tasks.

Event handling is implemented by listening asynchronously for events from all windows. Every window gets the correct event. Events can be processed by the window or converted into the configuration event in order to send to the application node [4].

4.5.4.4 Equalizer Server

The Equalizer server receives requests from all applications on the system and serves these requests using the application's specific configuration. It launches rendering clients on the nodes, determines the rendering tasks for a frame and synchronizes the completion of frames.

The server and the configuration are not application specific; these can be used in other applications [4].

Each configuration consists of two parts.

- The hierarchical resource description derived from a logical and physical environment of the application.

- The compound tree, which declares how the recourses are used in order to render.

The hierarchy is as follows:

[Nodes] > [Pipes] > [Windows] > [Channels]

Node: Represents a process. A node has pipes.

Pipe: Graphics boards in a machine. All pipes are executed in a separate thread. A pipe has windows.

Window: an OpenGL on-screen or off-screen drawable. All windows of a pipe share display lists and other OpenGL display objects. A window has channels.

Channel: Represents an OpenGL viewport in a window.

4.5.5 Summary

The parallel rendering approach is used in this study to display the output of the terrain rendering tool on multiple screens. The rendering detail and rendered area is increased by increasing the number of screens. After the survey of parallel rendering APIs, it is decided to use Equalizer for the synchronization of frames. Allowing the configuration of the number of monitors and GPUs dynamically is a great advantage of Equalizer on being elected as a frame synchronizer. The previous parallel rendering architectures are very complex to configure and manage except Equalizer which is described in 0. The other architectures work with pure OpenGL and they are not suitable for the system configuration changes. Equalizer has dynamically configurable architecture that works with many configurations like multi-pipe or multi-cpu. The only disadvantage for our case is that it works just with pure OpenGL. There should be a work to integrate this architecture with a scene graph mechanism. Scene graph mechanisms bring many advantages like culling or any other performance upgrades with it.

CHAPTER 5

AN ADVANCED TERRAIN EDITOR WITH MULTI-SCREEN VIEWER

The major objective of this study is to develop a terrain editor tool with multi-screen support. In this chapter, the implementation of the terrain editor tool and multi-screen terrain viewer are discussed. The terrain editor tool creates, manipulates and saves terrain and the multi-screen terrain viewer receives the results of the terrain viewer then displays them on multiple monitors.

5.1 The Terrain Editor

A real-time interactive terrain editor which has several features in order to create and render terrain data is implemented. The terrain editor enables user to create a new terrain, modify and paint it. Users can create terrains either from scratch or from GIS data. The terrain editor is a GIS integrated tool. Algorithms that are mentioned in Chapter 2 are used here for all terrain editing and texturing processes.

Additionally users have the ability to place objects on terrain. These objects can be buildings, vehicles or vegetations. The terrain editor can also build roads, rivers and lakes interactively. Whatever user adds on 3D scene is also added to GIS. A screen shot of the application is shown in Figure 22.

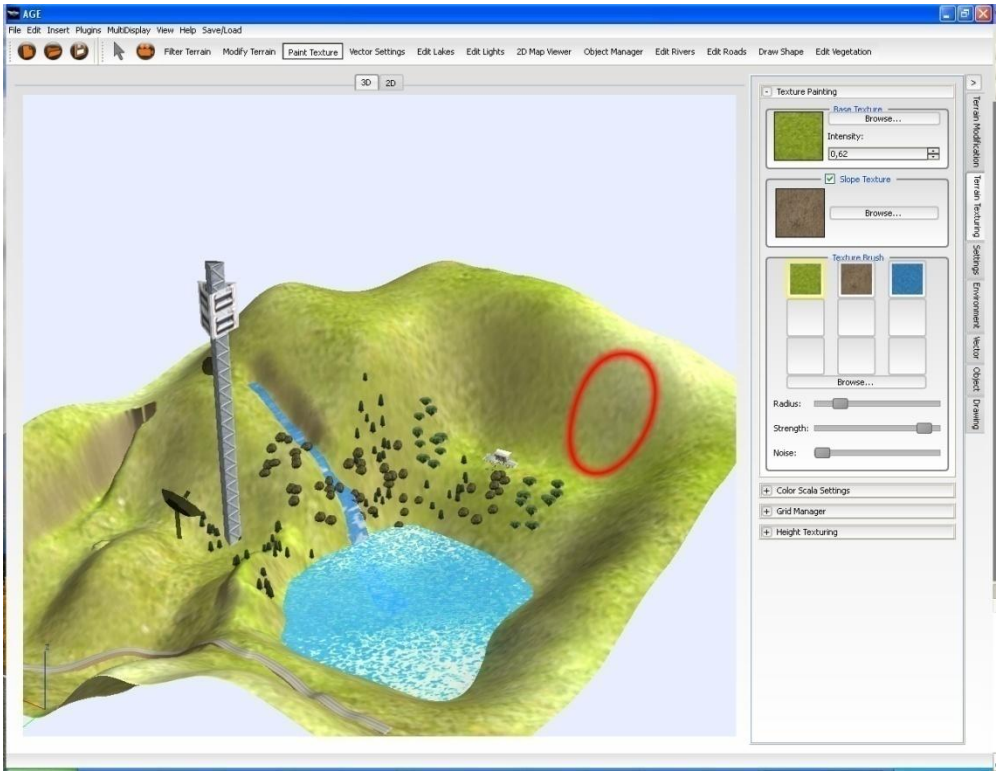

Figure 22 A screen shot of the terrain editor. A palette of the tool is placed on the right side, and the scene view is placed on the left side.

For future use, a plug-in mechanism that gives the application dynamism for scalability in its features has been created. Tool presents interfaces to plug-in to manage the 3D scene and GIS data.

The most of the features, such as placing objects or modifying terrain are also designed in the plug-in mechanism and given by default.

5.1.1 Terrain Editing

Terrain editing is the manipulation of terrain mesh which is controlled by the user of the application. There are several brushes which change the shape of a terrain or paint it. The most well known brushes are:

- Raise: Raises the vertices on the terrain mesh in a selected area.
- Lower: Lowers the vertices on the terrain mesh in a selected area.

- Flatten: Makes all vertices to the same level in a selected area.
- Smoothen: Makes the mesh in a selected area smoother.
- Add Noise: Adds random values to the z values of vertices on a mesh in a selected area.
- Paint Color: Paints a selected color on a selected area of terrain.
- Paint Texture: Paints a selected texture on a selected area of terrain.

There are several parameters that change the effects of brushes, such as radius, strength or noise. The effects of these brushes mostly change by the distance to the center point of the selected area.

At the implemented system, a plug-in mechanism is developed that enables any developer to create and add a new brush to the application. The application presents numerous services and interface to these services for the manipulation of terrains. These interfaces are implemented by plug-ins in order to use the services of this tool.

5.1.1.1 Possible Implementation Schemes

Several ways have been tried to edit terrain. These were:

- Sending height data as texture to the shaders
- Changing the primitive set data
- Editing vertices using the vertex buffer object

5.1.1.1.1. Sending Height Data as Texture to the Shaders

Sending height data as texture to the shaders is a simple method to apply in order to edit and render terrain. The height and the normal data are converted to textures and then sent to the shaders. Normal data is the vector that is perpendicular to surface at that vertex. In the shaders, the values of the height texture are used as the offset of the z value of vertices. Similarly, normal texture values are used as the normal values of vertices. In the editing process, it is planned that when the

user click on 3D scene for the modifications of the terrain, the color values of the textures is edited in the background. For each click on the terrain, the tool calculates height and normal values. Then these values will be set to the height and normal textures. After each process, changed textures will be sent to the shaders as whole and the results will be viewed interactively. Even if the idea is so simple, it does not work in reasonable frame times. It has been seen that sending the height and normal textures after every editing process is not an efficient way of editing. It becomes very inefficient especially when the editing of very little area of the terrain. The tool makes calculations about whole terrain heights and normals and sends two large textures even for a little change of the height data. The other disadvantage is depth of image. Since the color values are between 0 and 255 in an image it becomes impossible to send float values in a gray-scale image. Sample height fields can be seen in Figure 23. Black regions represent zero height and white regions represent the higher parts of terrain. Sending a colorful image increases the calculation both on CPU and GPU. Using a colorful image for the height information also causes a heavy data to transfer information to the shaders for each modification. The final disadvantage that is realized after the implementation is the culling problem. Since the vertex positions are modified on GPU, culling mechanism cannot understand whether that pixel is inside the view frustum or not. Culling mechanism works on CPU and it is applied before rendering process. Changing the vertex position after culling process gives contradictory results. So we have not chosen this approach.

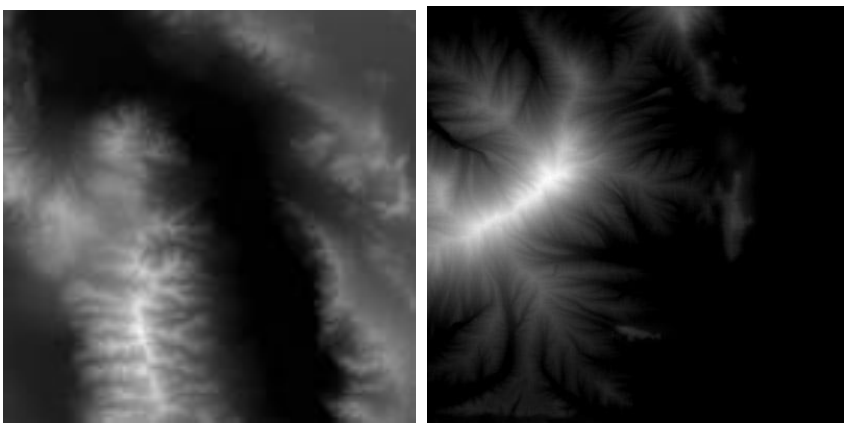

Figure 23 **Sample height fields. Lighter pixels represent higher regions and darker pixels represent lower regions.**

5.1.1.1.2. Changing the primitive set of data

Terrain geometries are composed of vertices of quads as in Figure 24. There are n-by-n vertices regularly spread on the scene and they are indexed in order to create a mesh of terrain with quads. On terrain creation, vertex and normal array and their indices are created. Then this geometry is created by adding a primitive set which draws these arrays of data. In editing process, it is planned to change the value of the vertex array. After each terrain manipulation process, vertex indices are calculated and then z values of these vertex data are changed but this change does not affect the mesh geometry on the scene. This geometry should be recompiled in order to see the effects of changes.

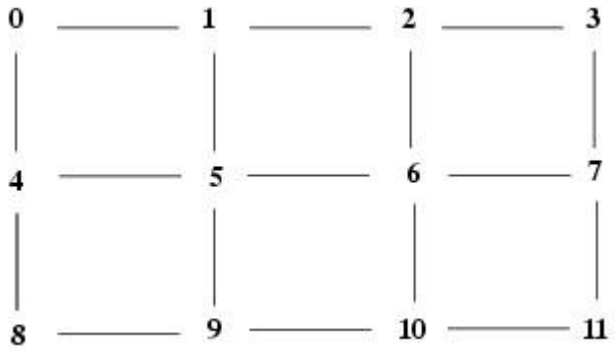

Figure 24 A terrain geometry that is composed of quads

After each modification, a flag of dirty display list has to be changed to true. This flag forces a recompile on next draw of any OpenGL display list associated with this geometry set when it is true. It means that after each modification, all terrain meshes will be created again. So this method also works slowly for interactive editing. After each editing process, the user should wait for a while even if changed area is so small.

5.1.1.1.3. Editing vertices using vertex buffer object

In the previous methods there is a big disadvantage of inefficient way of changing height data. Even for a little change of height data, all the terrain is calculated again or whole terrain data is sent to the shaders. That makes the user to wait for a while after each process.

This method uses vertex buffer object. A vertex buffer object is a feature that allows the tool to store the vertex data in the application memory [22]. Normally vertices are stored in GPU and they are not accessible from an application. VBO gives freedom to change the data of vertices on the application side. The structure of vertex buffer object is explained in Figure 25. This method enables accessing and changing the vertex data. On creation of terrain mesh, vertex data is buffered on CPU. After editing, the z value of vertex is changed [22].

To use this algorithm, the developer should change the flag to use display lists to false. Normally geometries are rendered by the display lists whose vertex values are static. Since the aim is editing of terrain geometry, the vertex values should not be static and they should be checked for each frame on editing.

The tool sets the values of changed vertices and after changing the height values, it calculates the normal values of changed parts. Since both the vertices and normals are mapped on memory, it is easy to set new values. Since this method does not change unnecessary vertices on editing, it works more efficiently then previous methods.

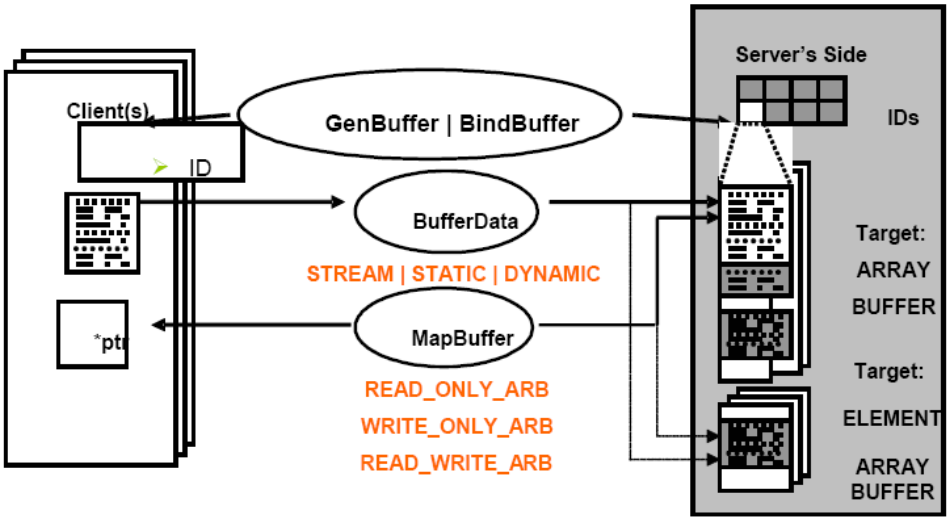

Figure 25 Using VBO [22]

5.1.1.2 Terrain Editing using Paging

Since the objective is to render very large terrains, paging method is preferred in this study. Paging is useful especially when there are many terrain tiles. For optimization, there is no need to render tiles which are far from the camera. Only tiles which are close to the camera will be rendered. In some paging methods, tiles which are far from the camera are rendered with low resolution.

In this editor, it is assumed that there is an editing window that stores 3-by-3 tiles. User can only edit the selected tile and its neighbors. This editing window can be seen from Figure 26. In this figure, red tile represents the selected tile and orange tiles represent neighbor tiles. Green tiles are not displayed. If the user wants to edit somewhere different from selected tile, he should change the desired tile as selected. If a selected tile changes, the tool automatically saves old terrain tiles and loads new selected terrain tile and its neighbors from the disk.

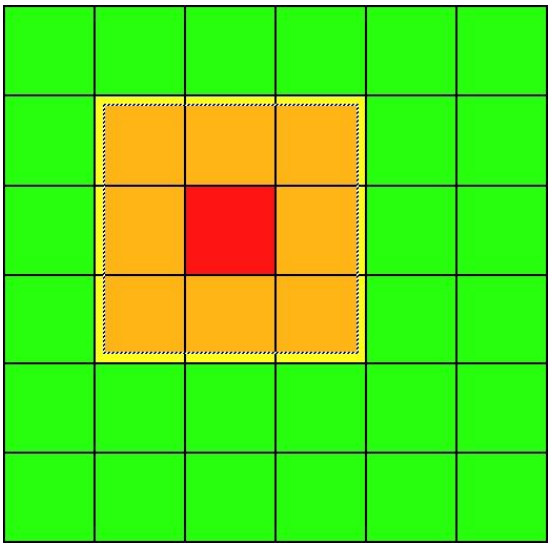

Figure 26 A sample diagram of 6x6 terrain pages and 3x3 editing window. Red square represents active tile, orange squares represent neighbor tiles and green squares represent the tiles which are not displayed.

On editing, a new height field is created that is composed of editable tiles. If the selected tile is not on the border of whole terrain, it is 3-by-3 terrain tiles. If somehow, the selected tile is on the border, it may be 2-by-3 or 3-by-2. If it on the corner of whole terrain it is 2-by-2 as in Figure 27.

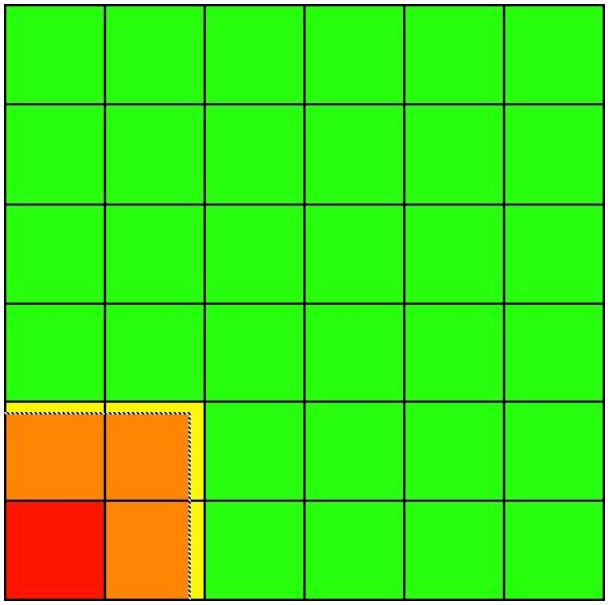

Figure 27 A sample diagram of 6x6 terrain pages and 2x2 editing window. When active tile moves to a corner of terrain, the number of neighbor tiles decreases.

On every change of active tile, a new height field is generated by the composition of this selected tile and its neighbors. Terrain tiles are created with this height field and the offsets of tiles on that height field. When the user edits terrain, the tool changes the height field on the background. After changing the height fields, the tool sends signals to the terrains to update their geometry by the new height values. If these changed parts are inside the boundary of that page, vertex buffer for height and normal values are updated. Using common height field has advantages on the border of terrain tiles and calculations related to neighbor vertex.

On changing selected tile, the tool does so many read processes from the disk. It is thought that there can be an optimization on loading tiles. If the new editing window and the old editing window have common terrain tiles, then there is no need to read data from disk for these common terrain tiles. One example of this case is shown in Figure 28. In this example center point moves to right and there are three common pages. The height maps of those common parts have taken from the old part and placed to the new editing window with an offset. Only the parts

that are not rendered on the old window are loaded from the disk. It decreases memory access. Since the most of the time, reading data from a disk is the bottle neck of real time systems, the performance is increased after this optimization.

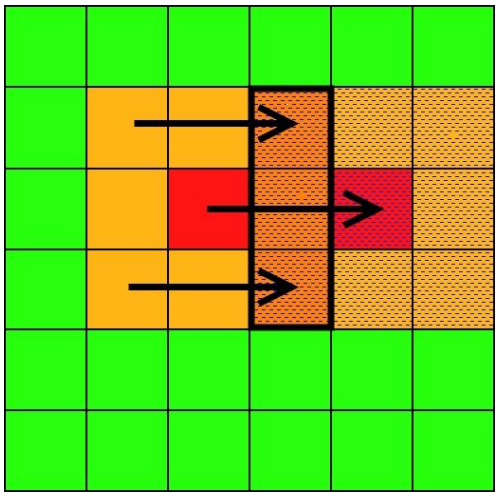

Figure 28 Common regions on editing window moves. The region which is outlined with bold line is not deleted. It is used for the next editing window.

One another optimization is that the terrain tiles are not saved, if the height field is not edited. On save process, the change of height field is checked. When a height field is loaded, a flag that shows if editing has done is set to false and this flag is changed to true if there is any editing process. On save process, this flag is controlled and related to the status of this flag terrain height field is saved to the disk. Similar to this process, terrain alpha maps that store texture which should be tiled, is written to disk, if it is changed.

5.1.2 Terrain Texturing

For terrain texturing, using very large texture is not efficient over the whole terrain. If you consider that a terrain tile is about 1024 meters x 1024 meters and each terrain has 256x256 texture resolution per meter^2, then there should be a texture which has a size 262144 x 262144. This is a very large image to load and save. It is so hard to store both in the disk and the memory of the GPU.

Instead of using that texture, a small texture can be used to tile to the all terrain. The problem here is that the same texture will be repeated on the whole terrain and there will not be any other textures on that terrain.

To handle all these problems, alpha maps are used in order to show the placement of tiling textures. Alpha maps have the same resolution as the height field of terrain tiles so one pixel corresponds to one meter square. And each one meter square area is textured with an image file that has at least 256x256 resolutions.

In alpha maps, red, green and blue channels of images are used in order to texture terrain. Red intensity shows the intensity of the first texture, similarly green shows the second and blue shows the third texture intensity. To increase the number of textures that are tiled on terrain, the number of alpha maps should be incremented. By default there are 3 alpha maps in the tool. Since each alpha map contains 3 textures, there will be 9 textures on the terrain. It is a satisfactory number of alpha maps and textures for a terrain editor. Since our texturing architecture is scalable, it is easy to increment the number of textures in future if demanded.

These three alpha map textures and the tile textures corresponding to channel of these alpha map textures are blended in shaders. An example of alpha maps is shown in Figure 29. In Figure 30, there are textures that correspond to red, green and blue channels of alpha maps. Result view is shown in Figure 31. Grass texture is tiled as “A” as the red parts in alpha maps. Similarly sand texture is tiled as the green parts in alpha maps which have the shape of “B”. Finally blue parts which have a “C” shape in alpha map is tiled with water textures. The mixture algorithm can be seen in Figure 32.

Figure 29 Alpha map texture. Each color represents one texture.

Figure 30 Tile Textures (Grass, sand and water textures from left to right)

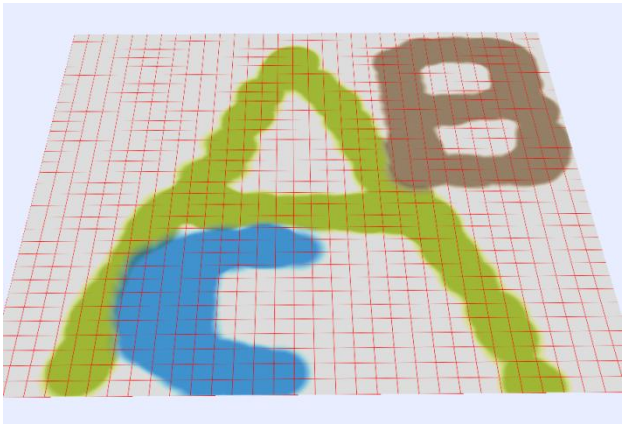

Figure 31 Result View (Red parts are covered with grass, green parts are covered with sand and blue parts are covered with water textures.)

```
varying vec2 texCoord;
varying vec2 texCoordAlpha;

vec4 mixTextures(
      in sampler2D textureR,
      in sampler2D textureG,
      in sampler2D textureB,
      in sampler2D alphaMap )
{
      vec4 textureColorR = texture2D(textureR, texCoordAlpha);
      vec4 textureColorG = texture2D(textureG, texCoordAlpha);
      vec4 textureColorB = texture2D(textureB, texCoordAlpha);

      vec4 alphaColor   = texture2D(alphaMap, texCoord);

      return (alphaColor.b * (textureColorB - 1) + 1)*
             (alphaColor.g * (textureColorG - 1) + 1)*
             (alphaColor.r * (textureColorR - 1) + 1);
}
```

Figure 32 The algorithm which calculates the mixture of three textures by alpha map texture in fragment program

Even if alpha maps are used for texture painting, there is still some repeating texturing on terrain tiles. These repeating textures can be seen from Figure 33 on the left side. If a region of one channel spreads on a large area, the texture is repeated on that region. To minimize this effect, a base texture is used. It mostly gives random noises on tiled textures so that repeated texture tiling will not be recognized by human eye. The results of this base texture can be seen in Figure 33 on the right picture. Instead of using a random image for base texture, a terrain related image that roughly shows the shape of terrain gives better results. A sample base texture is shown in Figure 34.

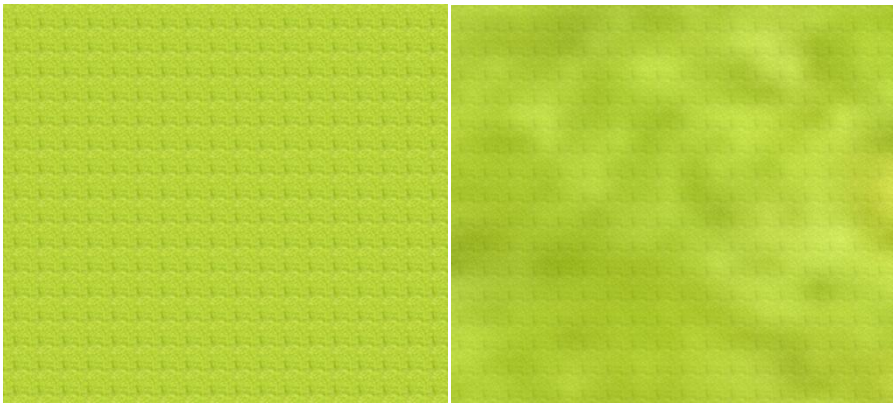

Figure 33 Terrain surface before base texture which is placed on the left and after base texture which is placed on the right. There is no tiling effect after using base texture.

Figure 34 Base Texture. A satellite image of Ankara is chosen for base texture in this example.

The intensity of the base texture is also adjustable. This intensity value varies from 0 to 1. When it is 0, non-painted parts are displayed as white and painted parts are displayed as its own texture as in Figure 38. If the value is 1, the tiled texture is not recognizable. Whole terrain is rendered with the base texture as in Figure 36. The algorithm of the base texture is given at Figure 35 below:

```
vec4 baseColor = texture2D(baseTexture, texCoord);

vec4 c1 = mixTextures(texture1, texture2, texture3, alphaMap1);
vec4 c2 = mixTextures(texture4, texture5, texture6, alphaMap2);
vec4 c3 = mixTextures(texture7, texture8, texture9, alphaMap3);

vec4 color = c1 * c2 * c3;

gl_FragColor = ( color * (1.0 - baseTextureIntensity) +
                 (baseColor * baseTextureIntensity) );
```

Figure 35 The mixture of base texture and tiled textures in fragment program

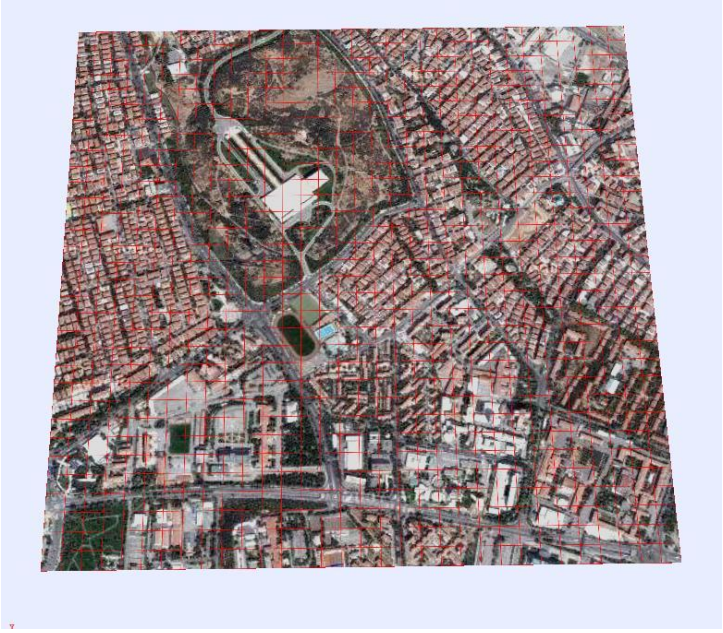

Figure 36 Base Texture with intensity value 1.0. Base texture is seen but tile textures are not seen.

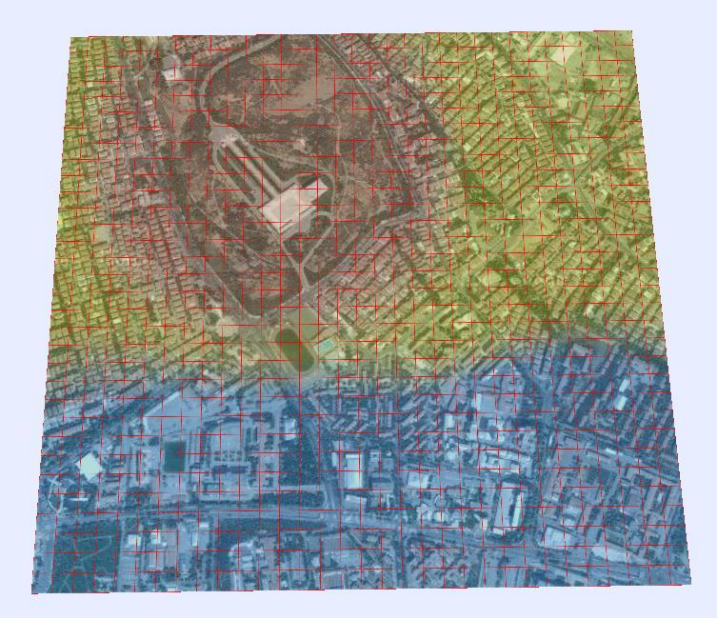

Figure 37 Base Texture with intensity value 0.5. The mixture of tiled textures and base texture is seen.

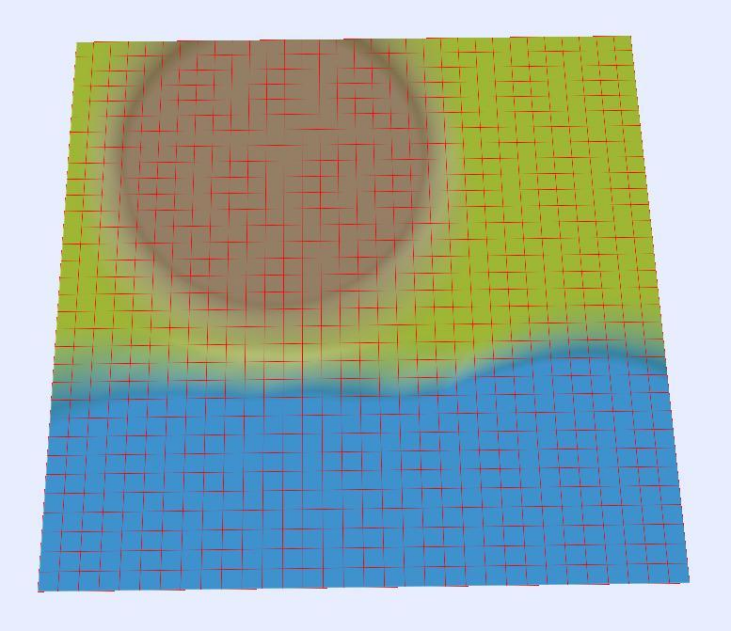

Figure 38- Base Texture with intensity value 0.0. Tiled textures are seen but base texture is not seen.

5.1.2.1 Texture Painting

On texture painting, textures are selected by the user. Since there are three alpha map textures, it is allowed to select maximum nine textures in order to paint terrain. The user may not select exactly nine textures. Nine is just the maximum number of textures to paint. On painting, user selects the active texture to paint and the parameters of the painting brush. These parameters are radius, strength and noise.

Radius represents the affected area of the brush from the center of the clicked point. Painting brush can also be square or circle. The change of radius is shown in Figure 39 and Figure 40.

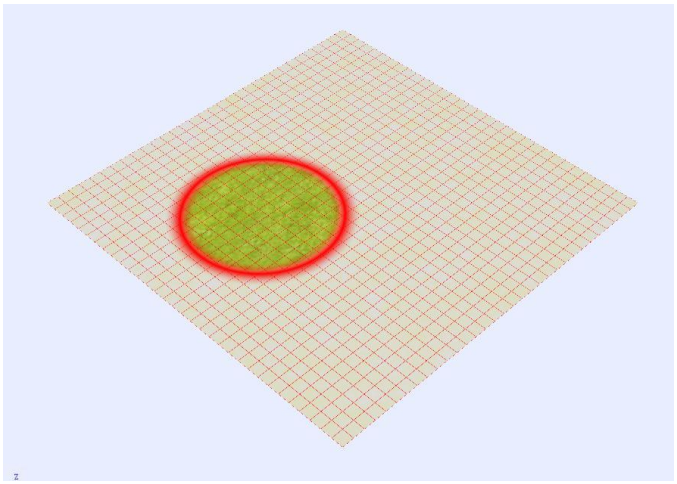

Figure 39 Texture painting with radius = 50

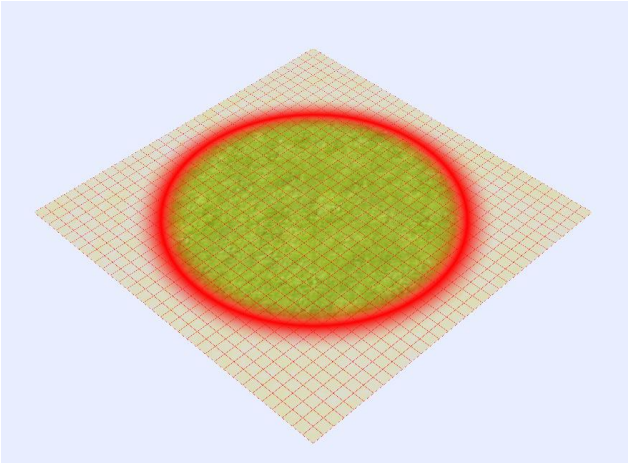

Figure 40- Texture painting with radius = 100

Strength represents the intensity of painted texture. It varies between 0 and 1. If it is a small number, user has to click too much to paint that area full of that texture as in Figure 41. If this value is one, whole selected area is directly painted with that texture as in Figure 42.

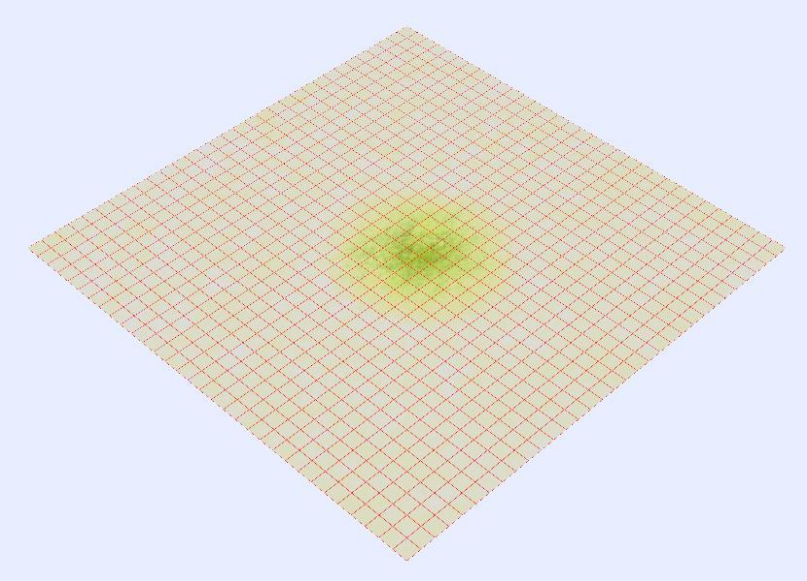

Figure 41- Texture painting with strength = 0.5

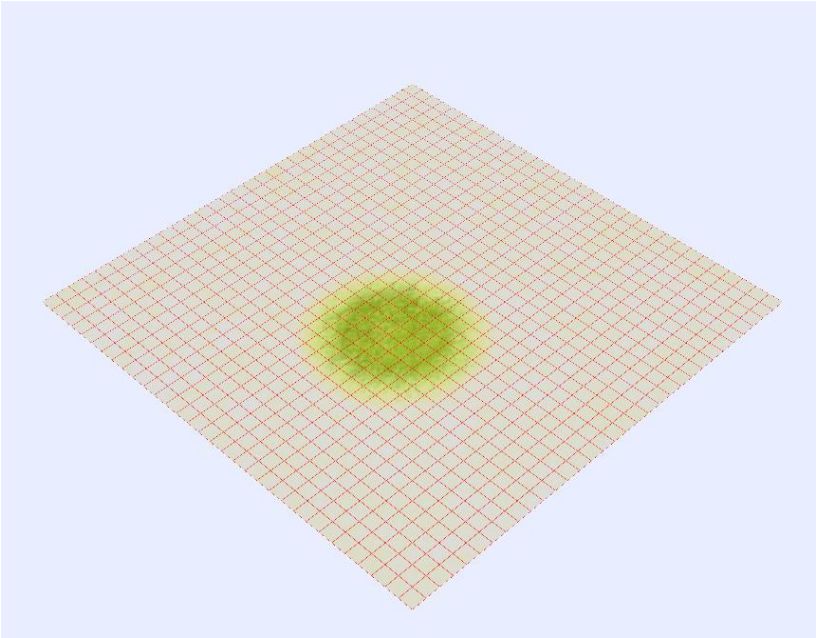

Figure 42- Texture painting with strength = 100

Noise value determines whether the user paints scattered or not. This value is between 0 and 1. 0 means there is no scattering and 1 means totally scattered on that area. The effect of noise is shown in Figure 43 and Figure 44.

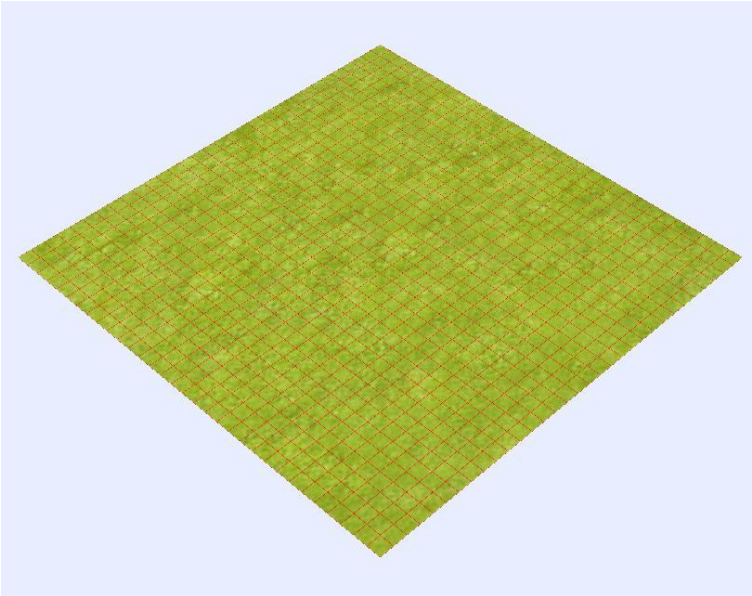

Figure 43- Texture painting with noise = %0

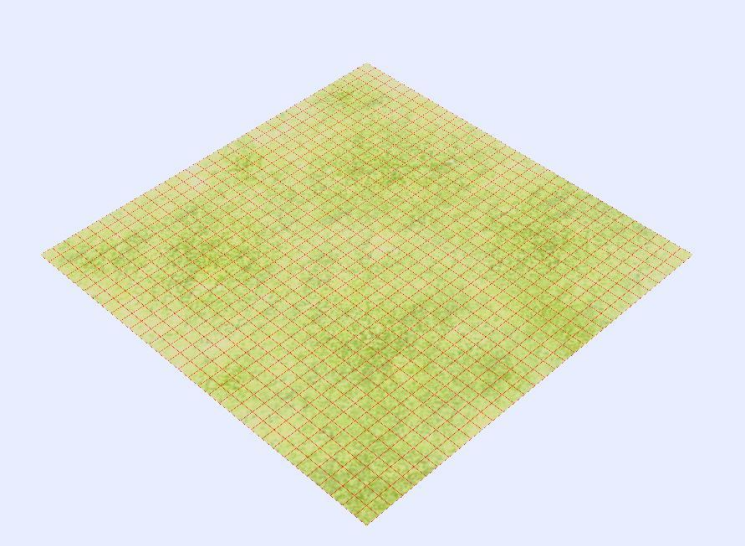

Figure 44- Texture painting with noise = %100

After selecting the texture and the parameters, user clicks on the terrain. With the real point on terrain and radius values, new pixel values are calculated. Affected terrains modify their alpha maps with these parameters. Each terrain tile manages its own alpha maps and sends new textures to shaders. This process is performed without decreasing the frame rates. Figure 45 and Figure 46 show the result of painting. In this painting process, images in Figure 47 are used for tiling textures, Figure 48 is used for alpha map and Figure 49 is used for base texture.

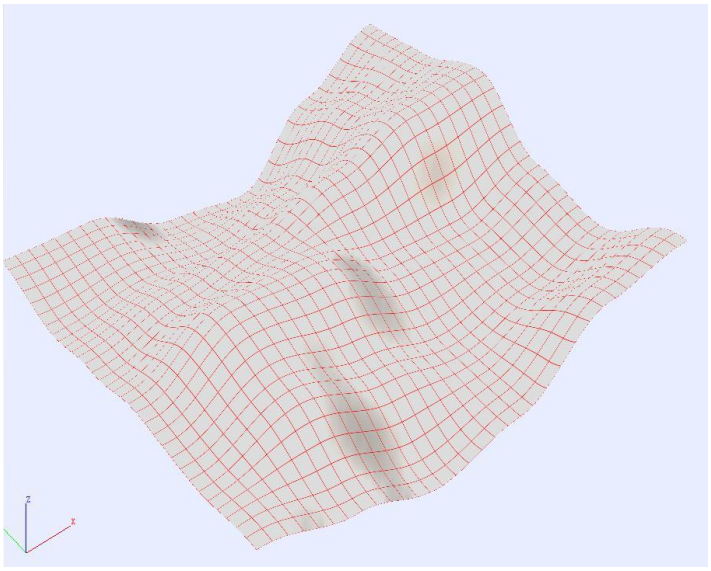

Figure 45 Terrain Before Painted. There are only grids on terrain.

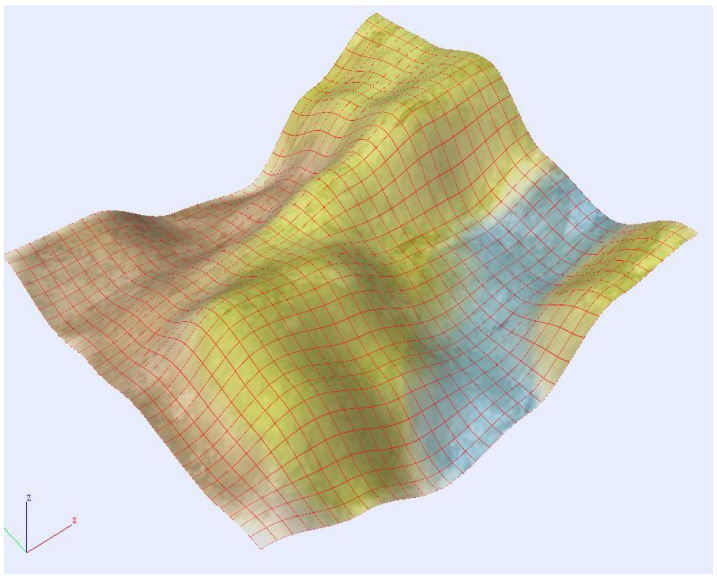

Figure 46 Terrain After Painted. There are textures and grids on terrain.

Figure 47 Tiled Textures

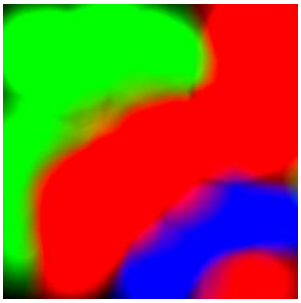

Figure 48 Alpha map texture

Figure 49 Base Texture

5.1.2.2 Slope Based Texturing

Slope based texturing is used on terrain regions where slope is close to 1 Generally rock or soil textures are used on those areas to display a realistic view. The slope is calculated by the normal value of vertices. *z* value of normal gives the slope value of this triangle. The calculation of aim is shown in Figure 50. Slope texture intensity increases by the value of normal on that pixel.

```
//#vertex program

uniform float size;

varying float slope;
varying vec2 texCoord;
varying vec2 texCoordAlpha;

//Since the values about geometry are sent to the vertex
shader, slope value is calculated in vertex program.

void main()
{
...

vec2 relatedCoord = gl_Vertex.xy - origin.xy;
texCoord.xy = relatedCoord / size;
texCoordAlpha = relatedCoord;

slope = gl_Normal.z; //z value of normal

...
}
```

Figure 50 The formula for the calculating slope of vertices on vertex program

```
//#fragment program

uniform sampler2D slopeTexture;

varying float slope;
varying vec2 texCoordAlpha;

void calculateSlopTexturing(inout vec4 pixelColor)
{
  vec4 slopeColor = texture2D(slopeTexture, texCoordAlpha);

  slope = clamp(slope, 0.0, 1.0);

  pixelColor = (1 - slope) * slopeColor + slope * pixelColor;
}
```

Figure 51 The calculation of intensity of slope texture on fragment program

As it is seen from the calculations in Figure 51, pixel color is changing according to slope texture value when the slope increases. If slope is 0, then pixel remains its own color. The results of slope texturing are shown in Figure 52 and Figure **53**.

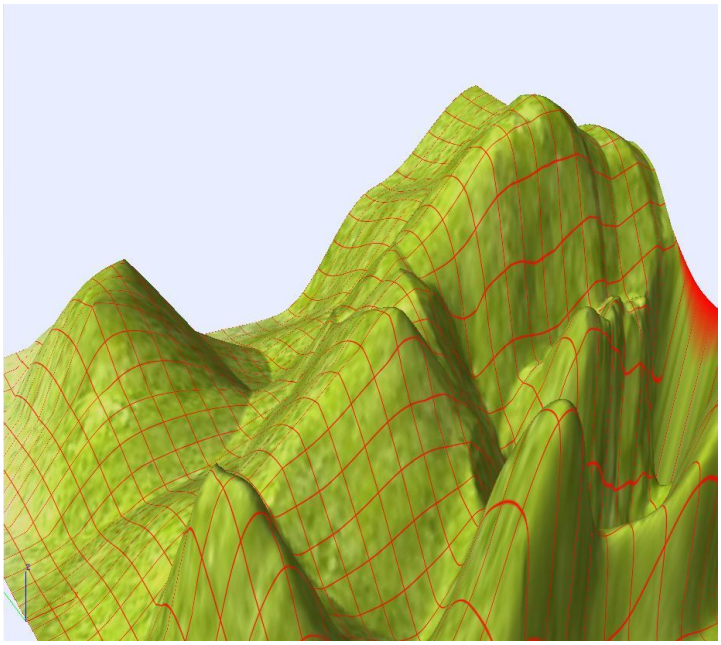

Figure 52 Terrain without slope texturing

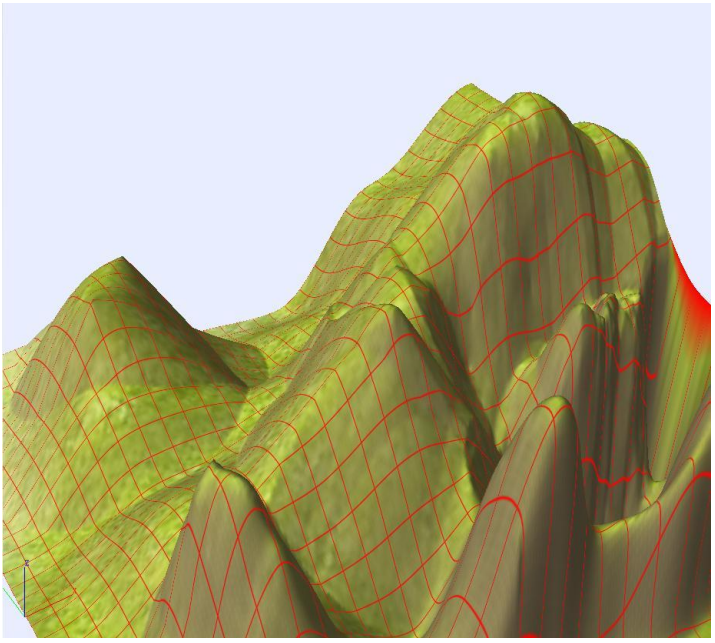

Figure 53 **Terrain after appliying slope texturing. Terrain pixels which has slope close to 1 are textured with soil texture.**

5.1.2.3 Procedural Texturing

In many terrain editors there is an automated texture of nature. In nature textures generally change with height. It starts with sea level which is blue, then vegetation comes which is green, and then sand comes which is brown and the most of the time there is a snow on the top of the mountains. The order may change to the regions or there are more levels in some regions in real life.

For that case, a plug-in which paints terrain automatically by height is implemented. Textures and height values correspond to these textures are selected. With these values, a new texture is generated on terrain with a single click. The results of the procedural textures are shown in Figure 54 and Figure 55.

To realize this idea, architecture is implemented on shader. Selected textures and their heights sent to the shader and the texture is calculated by the z value of vertex on rendering process. It is a fast way to implement, but the problem is that this painting process is not dynamic enough. Since terrain is rendered on shaders by checking just the z values of vertices, it is not possible to paint texture after this process. The user is forced to choose an option like either a terrain is textured with height values or terrain is textured manually by user.

This problem is solved by calculating height textures on CPU side. Height map values are processed and an alpha map texture is processed by the height textures and correspondent height values. This process takes a little time but it provides dynamism for the following parts of editing. After setting height textures, painting is still allowed, because this texturing mechanism is based on alpha map texturing. There is no need extra saving of the height values of height textures anymore by this method. Height textures are saved with the alpha maps.

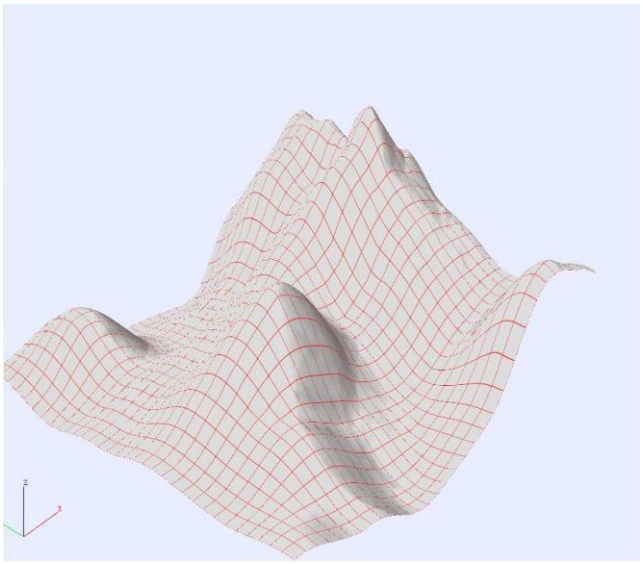

Figure 54 Terrain before applying height texturing

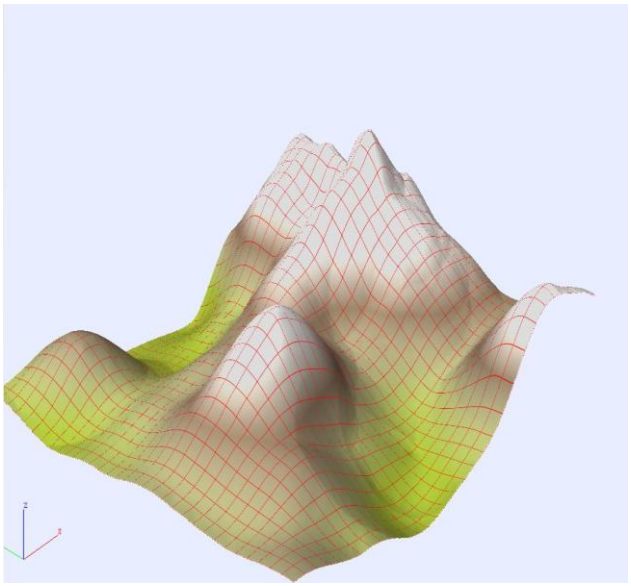

Figure 55 Terrain after applying height texturing. From lower to higher terrain is textured with grass, sand and snow textures.

5.1.2.4 Texturing with Satellite Image

Satellite images are taken photos from the satellites. If there is a real height data of the terrain, it is possible to use the photo of this place which is taken by a satellite instead of painting this area. It gives more realistic view on that terrain. An example of terrain which is tiled with a satellite texture is shown in Figure 56. A

satellite texture and the borders of this image are selected to apply on terrain. This process implemented on GPU side. Shaders take these textures and parameters to render that scene. Texturing with satellite image is applied after coloring with alpha map textures and slope texturing. Since satellite image has to override other paintings, this should come last.

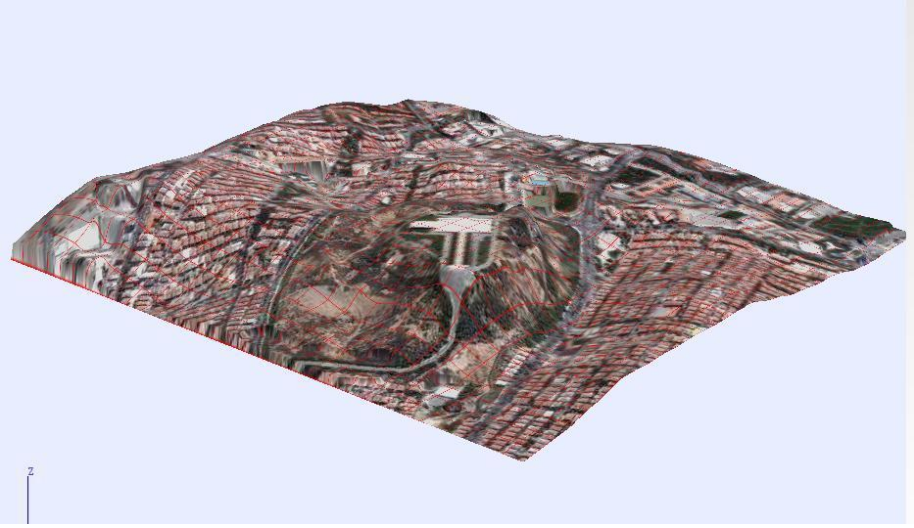

Figure 56 Sample Texturing with Satellite Image

5.1.2.5 Lightmap Texturing

A light calculation is sometimes very expensive process especially if there are more than 8 lights on the scene. Some algorithms are developed to solve this problem but the most suitable one for terrain is using a pre-calculated lightmap. Every light on a terrain is preprocessed and calculated. The results of these calculations are written on a lightmap that shows where should be illuminated and how much it should illuminated as in Figure **57**. The created alpha map resolution is same as the size of terrain grid. It can be increased but this resolution gives satisfactory results.

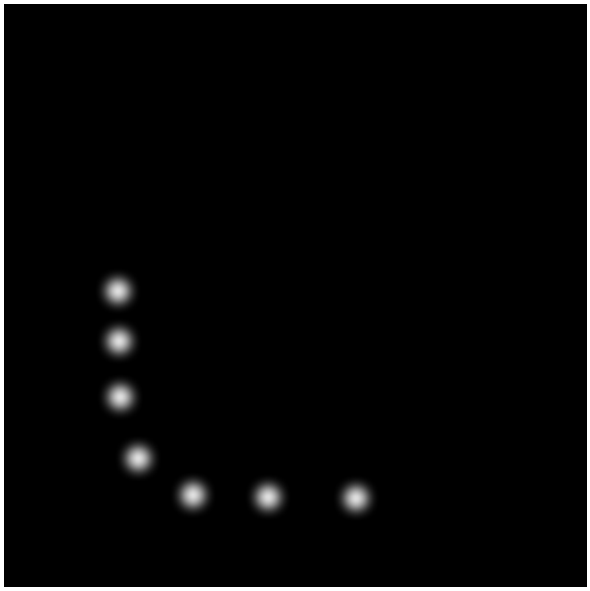

Figure 57 **Lightmap Image. White Parts represents illuminated parts and black parts represent dark parts.**

Lightmap process is implemented on GPU side. Lightmap texture is sent to the shaders that calculate lighting. Real light value and pre-calculated light value are added. The illuminations of lights which come from small light sources may not be recognized by human eyes. If pre-calculated light is directly added to real light value, the illuminations of these small light sources become recognizable. Actually it is an illusion of human eye. The pupil of eye expands in darkness and shrinks under light. Since it shrinks in daylight, the illuminations of small light sources are not recognized by us. To give this effect, light values that come from lightmap texture are decreased on day light and increased on night. A soft transition is implemented for the lightmap of day and night which can be seen in Figure 58, Figure 59 and Figure 60. The lightmap calculation is shown in Figure 61.

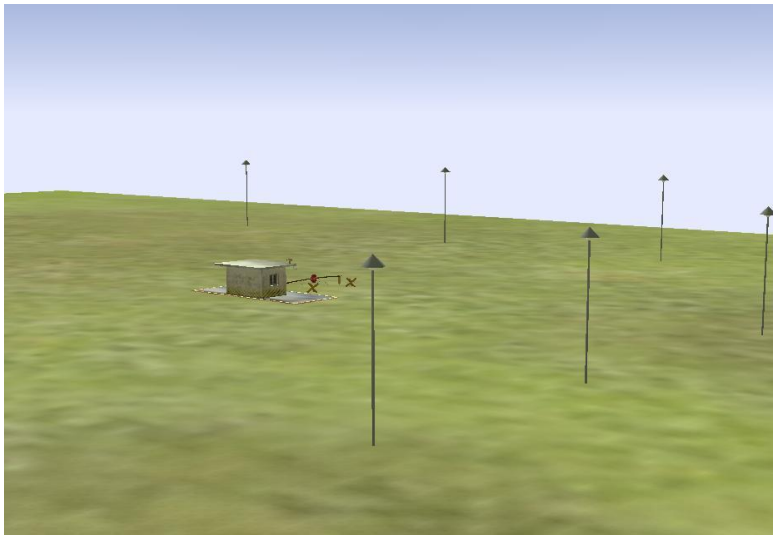

Figure 58 Screenshot from day time

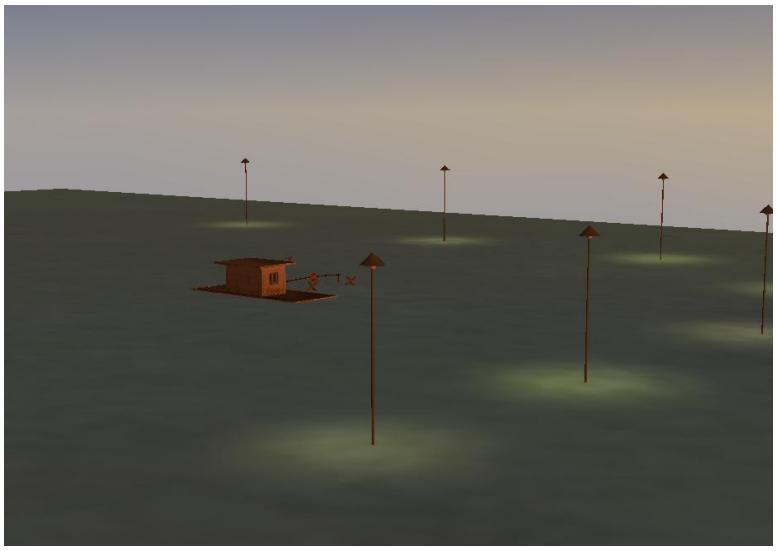

Figure 59 Screenshot from sunset

Figure 60 Screenshot from night time

```
//#lightmap shader

uniform sampler2D lightmapTexture;

varying vec2 texCoord;

//real light value calculated in vertex program
varying float lightAmount;

void calculateLightmap(inout vec4 pixelColor)
{
  vec4 lightColor = texture2D(lightmapTexture, texCoord);
  float lmIntensity = 1 - gl_LightSource[0].diffuse.g / 3;

  vec4 color = (lightAmount * pixelColor);
  color = clamp(color, 0.0, 1.0);

  pixelColor = (lightColor * pixelColor) * lmIntensity + color;
}
```

Figure 61 The lightmap calculation in fragment program

5.1.2.6 Other Colorings

In this part, the processes which are not for a realistic rendering of terrain are discussed. These processes help user to edit and analyze the terrain geometry. The information is mostly given by coloring some regions of the terrain.

5.1.3 Coloring by Height

Coloring by height looks like a geographical map. User can decide any color for any height. Since there will not be any painting after that process, it is implemented in GPU side. These values are sent to the shaders. After sending these values, it is directly displayed on the screen without any latency. Original terrain is shown in Figure 62 and the terrain after the coloring by height process is shown in Figure 63. Coloring is done by the values in Table 2.

This process is so similar to texturing by height, but the only difference is that it is applied on GPU. This coloring is not only for realistic viewing but also for understanding height values.

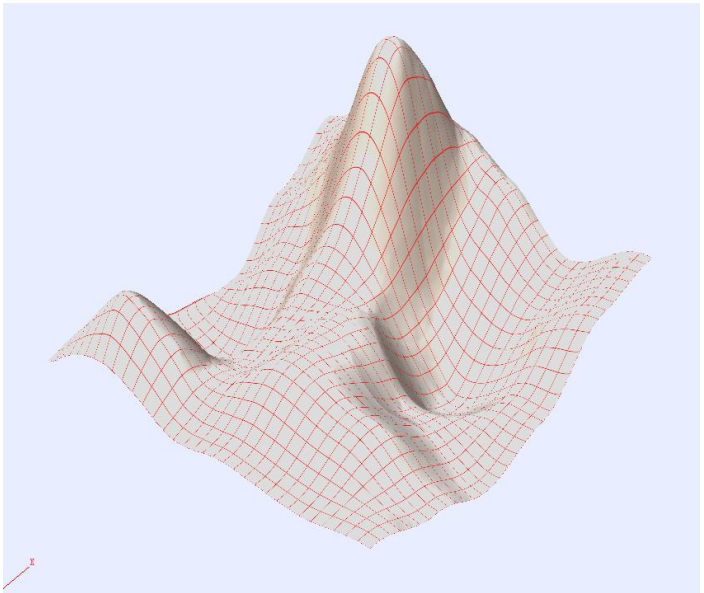

Figure 62 Terrain before elevation texturing

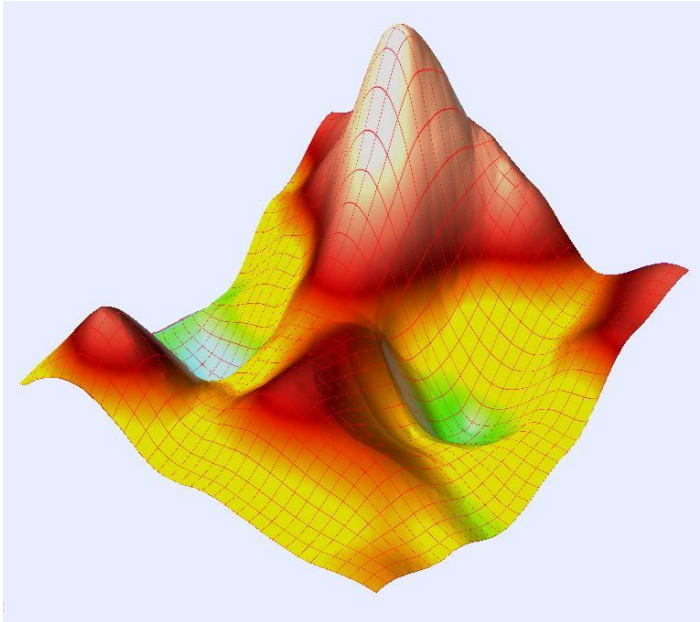

Figure 63 Terrain after elevation texturing. Each height is represented with different color.

Height Values	*Colors Values (Red, Green, Blue)*	*Color*
0	0, 0, 255	
33	85, 170, 255	
66	85, 255, 0	
99	255, 170, 0	
132	85, 0, 0	
200	255, 170, 127	

Table 2 Height values and correspondent color values

5.1.4 Terrain Grids

Measuring is always done by referencing of a constant field. If the terrain information is not known before, the size of terrain cannot be realized. 256m x 256m terrain, looking from 1000m distance has the same look 1024m x 1024m terrain looking from 4000m distance. In real life there are always references, such as buildings, trees, plants or any other surface texture. In computer graphics there is always a limit of the resolution.

Terrain grid is implemented in order to help users to understand terrain size or the distance between two points. Lines are drawn in horizontal and vertical with a constant space as in Figure 64. This process is also implemented in GPU side. To draw lines, it is checking x and y values of vertices. After all painting process it shows terrain grids.

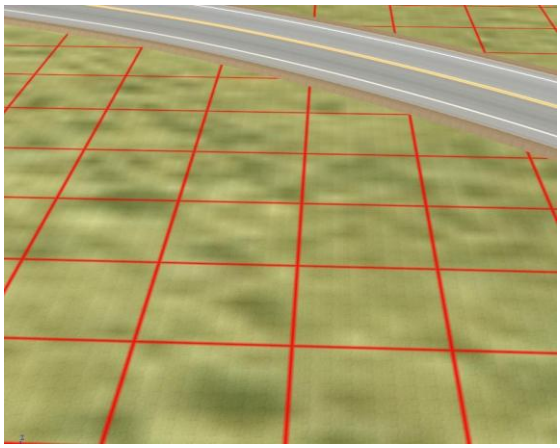

Figure 64 A screenshot of a grid (Each line has 8m distance.)

5.1.5 Contour Lines

In cartography, contour line is the connection of points which have the same height from the sea level. It also helps user to understand the level of terrain as in Figure 65. This process is also implemented on GPU. To draw contour lines, it is checking the z values of vertices and the distance of each contour line.

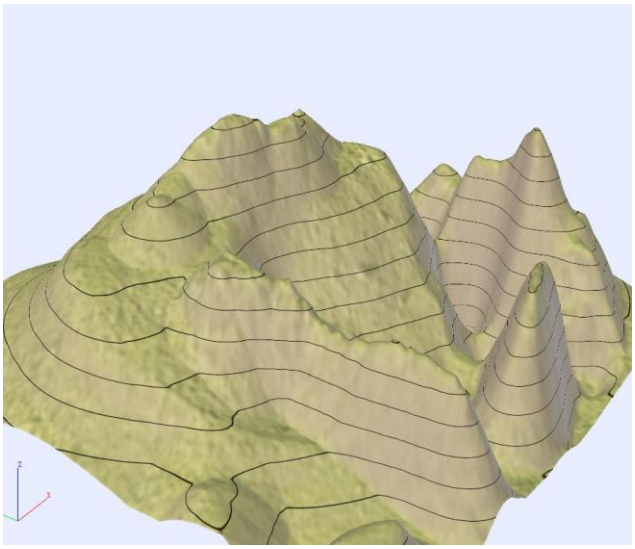

Figure 65 A screenshot of contour lines (Each contour line has 10m distance.)

5.1.6 Editing Circle

In the most of the editing process, users need to know where they are editing and how much area this process will affect. To accomplish this need, an editing circle is used (See Figure 66). This layer is implemented in GPU side. Tool sends radius and mouse pick coordinate to shaders and after all painting process, shaders render that circle to show editing area.

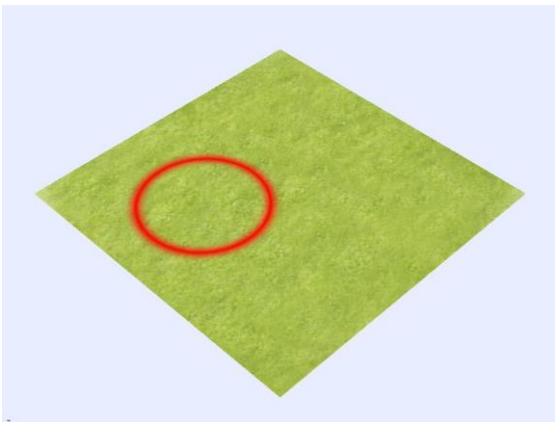

Figure 66 Editing Circle (Radius = 50m)

5.1.7 Editing Square

Similar to editing circle, an editing square is implemented. Everything is the same but the shape (See Figure 67).

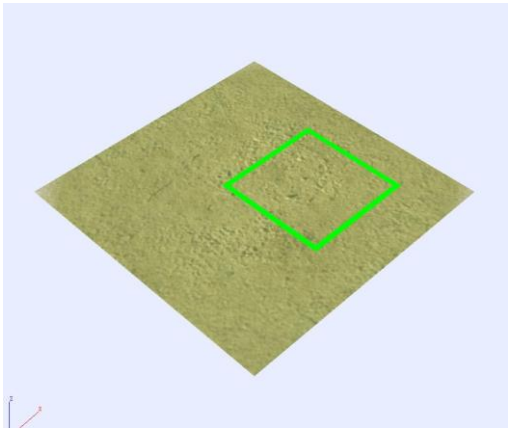

Figure 67 Editing Square (Length = 100)

5.2 The Terrain Viewer with Multi-Screen Support

The terrain viewer is implemented in order to render very large terrains which are created by the terrain editor tool. The terrain viewer loads the terrain from disk in pages and calculates a level of detail for each page. These pages are rendered in different levels of detail according to the distance to viewing camera. Currently there is no editing feature available on viewer. This tool is mainly designed in order to view the results of the terrain editor. After creating a terrain with the editor, the terrain viewer tool is called in order to view all the pages of the terrain. The viewer reads terrain data in pages and loads to the scene. The whole terrain is rendered altogether with different levels of detail. The first time that a user is able to see whole terrain pages together is in that viewer.

When user moves towards terrain, new pages are loaded in threads. The resolution of pages increases when the distance of camera to the rendered page decreases and vice versa the low detailed pages are loaded when the camera distance to this page is increased.

On viewing the terrain, user has also an option to view scene in multi-screen as in Figure 68. In this example, there are three monitors which are connected to different graphics cards. The application checks the configuration files to decide the number of screens, the point of view and the field of view of cameras. All these values are stored in the configuration files.

Figure 68 A photo of the multi-screen system with three monitors

To increase the performance of the system, the main configuration uses multiple application nodes. This configuration works better on computers which have multiple graphics boards. The calculations of vertices and textures are done in different graphics boards so it gives parallelism in calculations and rendering. By this way, the screen resolution is incremented with very little loss of fps or no loss.

This tool can also work on different computers. Suppose there are *N* computers, each has *M* monitors. This tool lets users to view scenes with *N x M* monitors which show that our system is scalable on resolution.

In the following sections, the infrastructure of the multi-screen terrain viewer is explained. Terrain paging with level of detail support is explained in section 5.2.1 and the implementation of parallel rendering is mentioned in section 5.2.2.

5.2.1 Paged Terrain LOD

Since computers have limited resources, rendering very large areas is a very challenging task. Page mechanisms are created to reduce the load on the computer. Whole terrain is divided into pages and when these pages are placed in an order, whole terrain is generated again. The computer does not load whole

pages at the same time, but just loads the pages that are close to the view point. The number of pages that are loaded is dependent on computer resources, such as memory, CPU and GPU power. There are two ways of incrementing this number of loaded pages. The first one is upgrading the computer to a super computer that has many GPUs and CPUs. The other way of increment is optimizing the scene to the computer. The most important optimization in rendering is "Level of Detail". This LOD mechanism renders terrain surface in details if this place is close to the view point. Likely the terrain surface is rendered in fewer details if this surface is far to the viewing camera. The surface area that is rendered at one view is incremented by paging and LOD mechanisms.

To this purpose, several ways are implemented and tested to combine "Paging" and "Level of Detail".

These are:

- Simplification Algorithm
- Tiled Block Methods
    - Paged Tile Blocks
    - Merged Paged Tile Blocks
- Threaded Loading Simplified Pages

5.2.1.1 Simplification Algorithm

In this method, a simplification process is applied after the terrain is created. Simplified terrains are created and saved to disk in several detail levels. These levels are up to user decision on creation. This simplification process can create either irregular or regular grids. These simplified terrains are good representations of real terrains because they are simplified by the elimination of vertices whose absence does not recognized. Regular grid and its simplified version are shown in Figure 69 and Figure 70. This algorithm creates a non-regular grid.

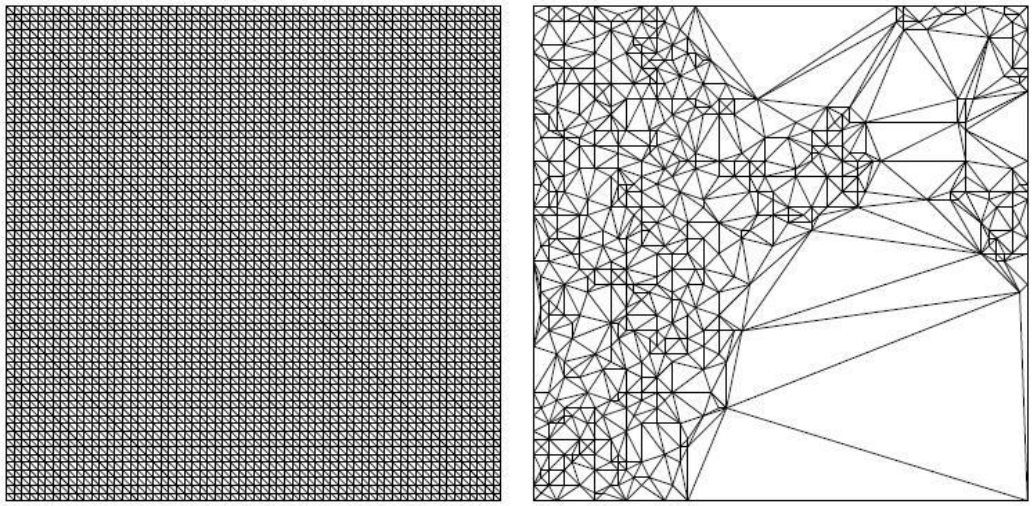

Figure 69 A top view of a regular grid (on the left) and its simplified version (on the right) [24]. The simplified version has fewer triangles.

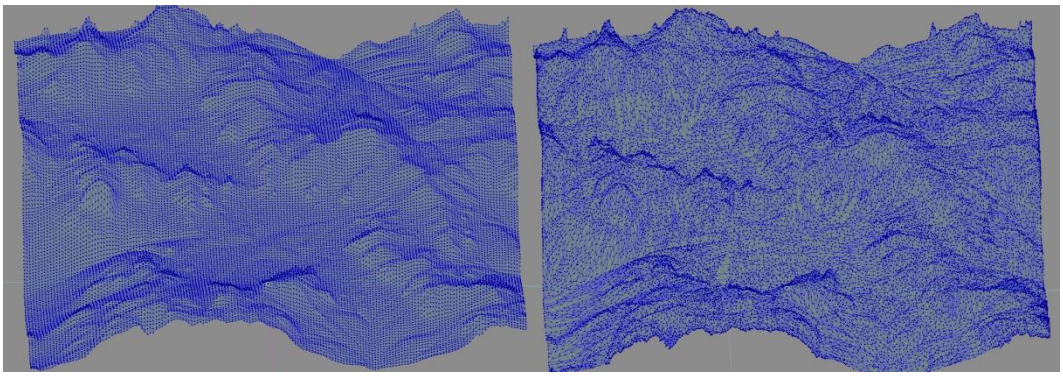

Figure 70 A top view of a regular grid (on the left) and the simplified version (on the right) [25]. The simplified version has fewer triangles.

Although this method is good at imitating the real terrain, this algorithm has many disadvantages. A considerable time is required in order to create the levels of original terrain. Not only the creation of these low resolution terrains takes long time, but also the new terrains with low resolutions may take up 3 or 4 times bigger size on disk than the original ones do. Since the original terrains are regular meshes, the height values are stored in a float array format in a binary file but the irregular meshes should be stored as in an array of triangles. Since each element of vertices has 3 values as x, y, z, the size of new file should be larger than the original terrain file. The reading process of these large sized terrains is also a big

job for a computer at interactive frame rates. Management and editing these terrains is hard too. Once you create a terrain, it cannot be modified. If a change is desired, it has to be built again. So this algorithm is not used.

5.2.1.2 Tiled Block Methods

Tiled Block method is mentioned in the related work part of chapter 2. This method is mentioned for a single paged terrain in the previous parts. Since our terrain is paged, this method should be modified to this case.

5.2.1.2.1. Paged Tiled Blocks

In this method, each page is rendered in separate LOD mechanisms. These pages are placed in the scene with their offset to the reference point. The tool does not care of relation between pages. In this algorithm whole terrain data is loaded for each page and vertices are eliminated when a simplification needed related to distance to the camera. In every frame time, this discarding calculation is done when the camera moves. Whole terrain page geometry is generated by the camera distance.

After applied this method to the terrain, it has seen that this algorithm does not work as fast as it is expected. All terrains become high resolution and frame time decreases dramatically, when the camera is close to the center of pages. Large cracks between terrain pages are also occurred on rendering terrain. Then it is decided not to use this method because of these large cracks and low frame rates.

5.2.1.2.2. Merged Paged Tiled Blocks

Merged paged tiled blocks method is similar to the Paged Tiled Blocks but the difference is pages are merged. Before applying tiled blocks algorithm, whole terrain pages are merged in order to obtain one page. Whole height map and textures are merged at the beginning of the application. Then the whole terrain is rendered in tiled blocks algorithm. Since whole terrain is merged, there are no

cracks on this algorithm but it did not suit the soul of paging mechanism. There is no paging on rendering in this algorithm. Paging is just used to store the terrain.

As a result this method is not used because it needs very long preprocess and it has not satisfied expected frame rates.

5.2.1.3 Threaded Loading Simplified Pages

In the previous methods, there has not been dynamism on loading pages and on rendering, frame rates were unsatisfactory. The method for this program has to dynamically load terrain pages when needed and remove pages which are not close to the camera. Frame rates should not be affected when loading terrains. To this purpose, a thread mechanism that loads terrain pages without decreasing frame rates is developed.

Another difference in this algorithm is whole page is rendered in one level of detail. In previous methods, pages were simplified partially related to their distance to the camera. In this method, every page decides if the level of detail should change and loads new level in thread when needed. When thread finishes loading new level, old terrain page is removed from the scene and new level is added to scene. Pseudo code of this process is shown in Figure 71. This process causes very fast frame rates that a human eye cannot recognize.

```
• Send the camera position to pages
• Each page creates a thread that calculates new
  resolution
    – If the resolution should be increased
        • Read data from disk
    – If the resolution should be decreased
        • Simplify the old page
    – If new resolution is changed
        • Generate a new terrain
• If the new terrain is generated
    – Load the new terrain
```

Figure 71 Pseudo code of thread usage on changing the resolution of terrains

This method is faster than the simplification algorithm in the previous parts because there is no calculation. The algorithm reduces the resolution by half by discarding the half of vertices as Figure 72. It discards vertex rows and columns regularly by one skipping. The original terrain page is shown in Figure 73 and low resolution versions are in Figure 74.

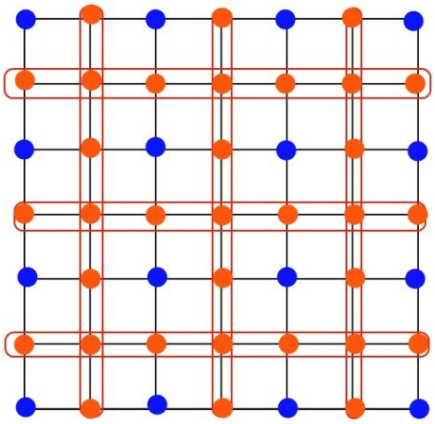

Figure 72 **The diagram shows how the elimination is done. Blue dots represents, remaining vertices. Orange dots are discarding vertices.**

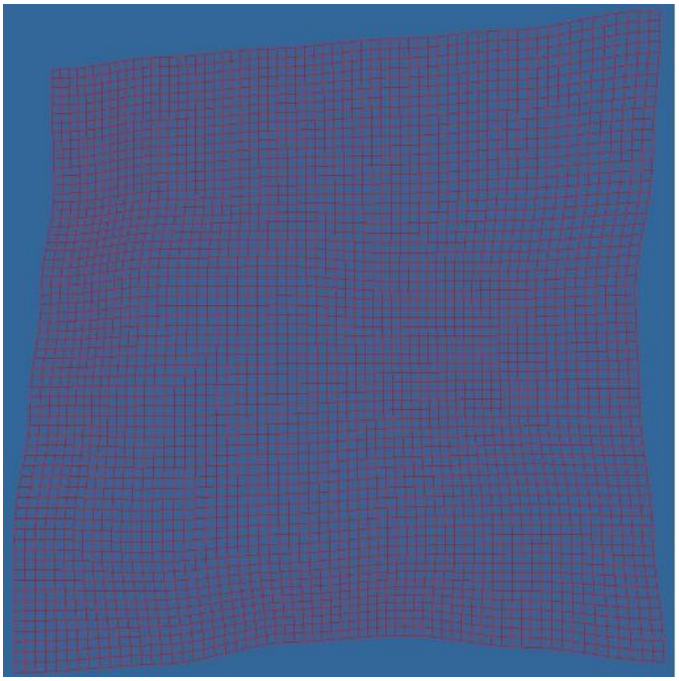

Figure 73 Terrain page without LOD

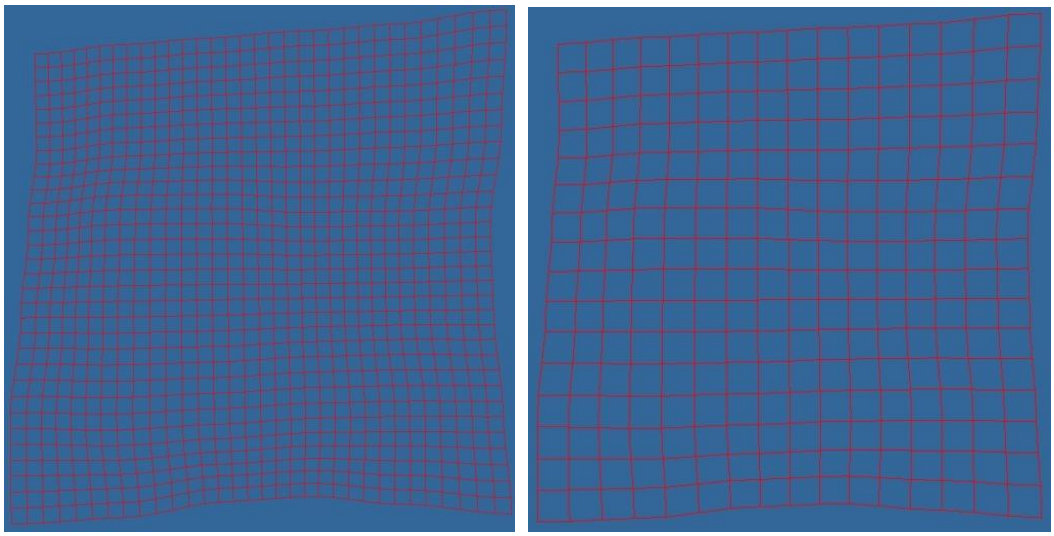

Figure 74 Terrain Pages with LOD = 1 (on the left) and LOD = 2 (on the right). If LOD increases, the resolution decreases.

The other advantage of this method is; it loads new pages dynamically. In previous methods, the sizes of terrains are constant. This method brings scalability on terrain loading. By managing the LOD levels and the computer hardware, it is

easy to see more terrain pages at one frame. It can be optimized to current computers.

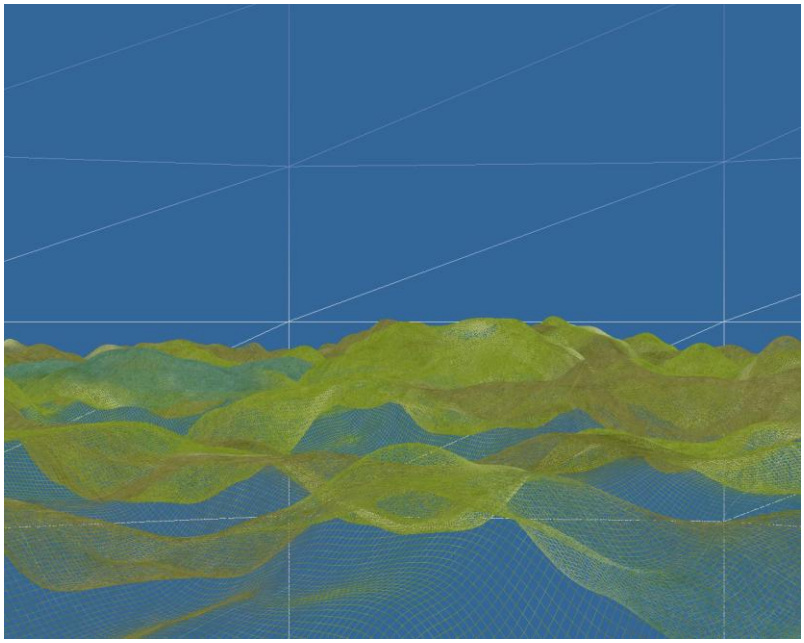

Figure 75 Terrain Geometry without LOD. All the polygons are in the same size.

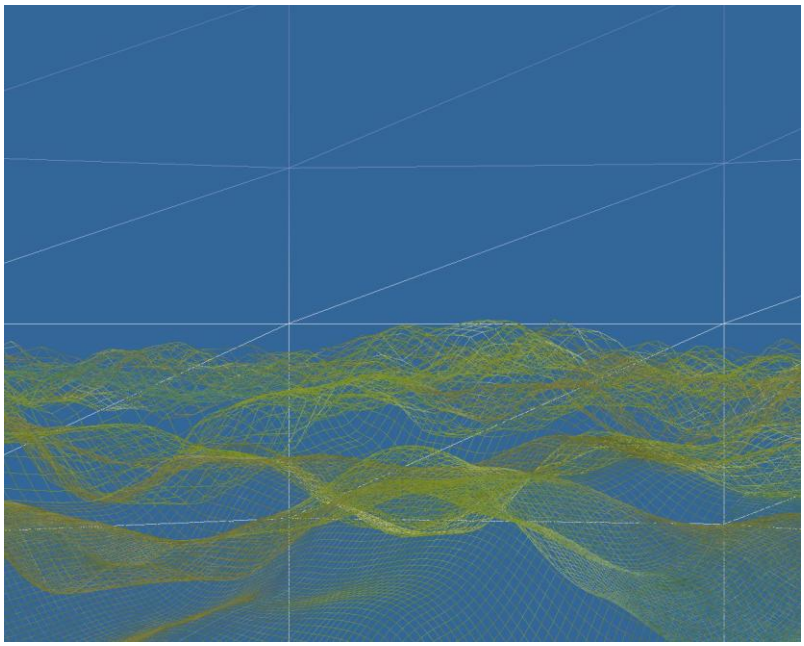

Figure 76 Terrain Geometry with LOD. Farther regions are represented with fewer polygons and closer regions are represented with more polygons.

This algorithm solves the performance bottleneck caused by the number of vertices. Terrains are loaded very coarsely if they are far away from the camera (See Figure 75 and Figure 76). Even if these terrains are very low polygonal meshes, they are tiled with full resolution textures. This is our next bottleneck.

To solve this problem, a texture LOD mechanism is applied. If a terrain is further than a constant distance, tiling textures are changed just with an average color of that tiling texture and base textures are loaded with low resolution replicas (See Figure 77 and Figure 78). This method solves the high memory consumption problem and fastens the loading terrains.

The performance improvement of LOD mechanism can be seen from Table 3 and Figure 79.

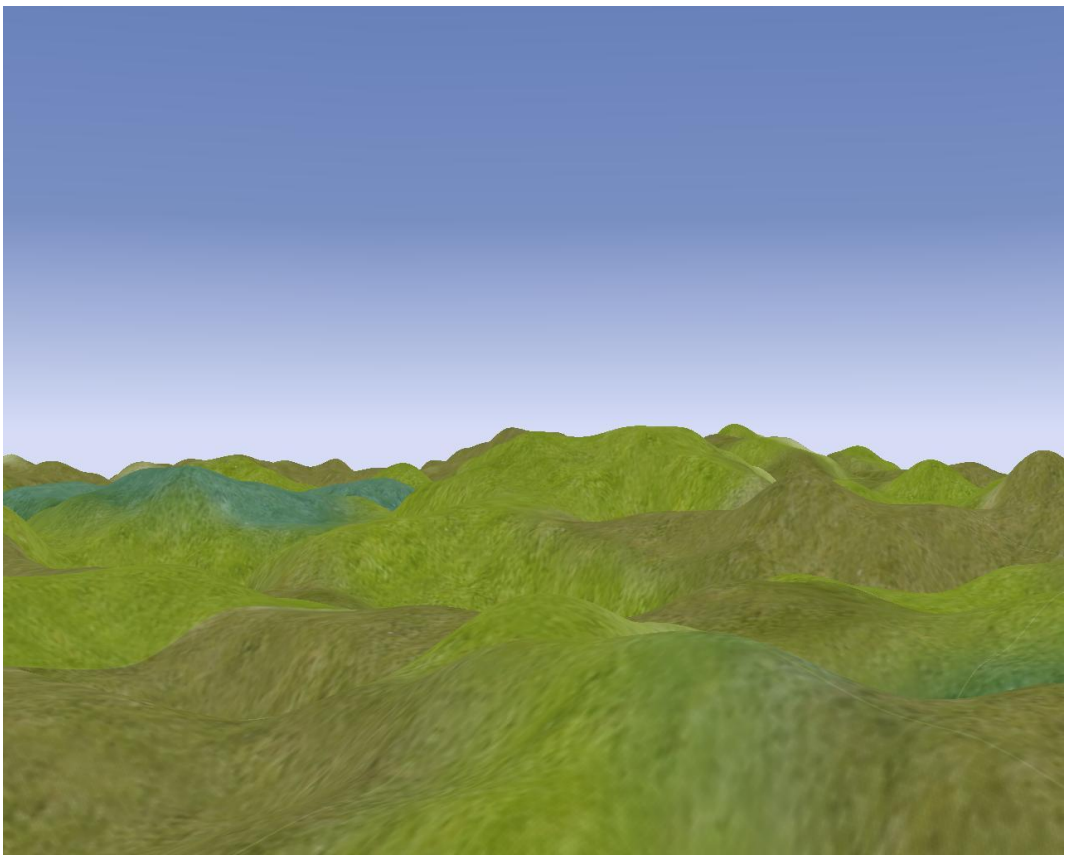

Figure 77 Terrain texture and geometry without LOD

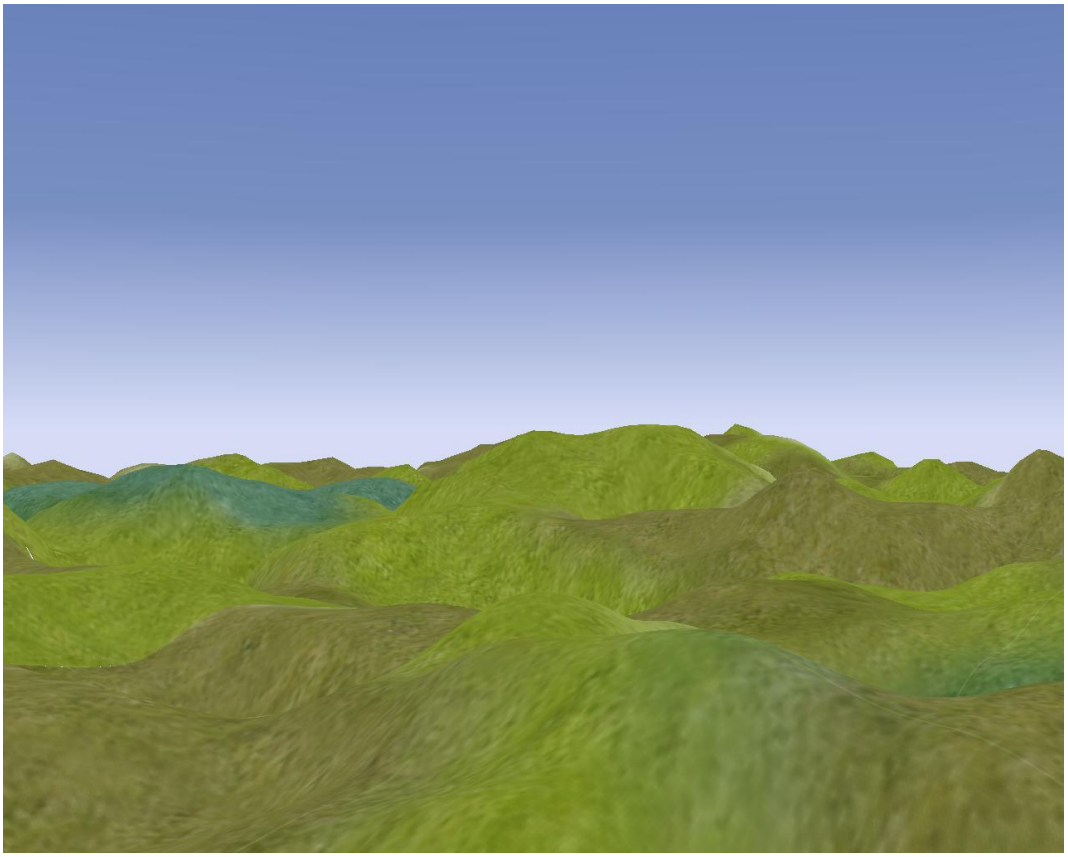

Figure 78 Terrain texture and geometry with LOD

Terrain Size	Number of Pages	Without LOD (FPS)	With LOD (FPS)
1024 x 1024	4 x 4	232	352
2048 x 2048	8 x 8	105	305
4096 x 4096	16 x 16	51	150
6144 x 6144	24 x 24	28	68
6400 x 6400	25 x 25	16	58
6656 x 6656	26 x 26	N/A	57
7168 x 7768	28 x 28	N/A	53
8192 x 8192	32 x 32	N/A	48

Table 3 A performance comparison table of with LOD and without LOD algorithms

(Each page has 256x256 vertices.

N/A areas could not be measured because the tool exits with out of memory error.)

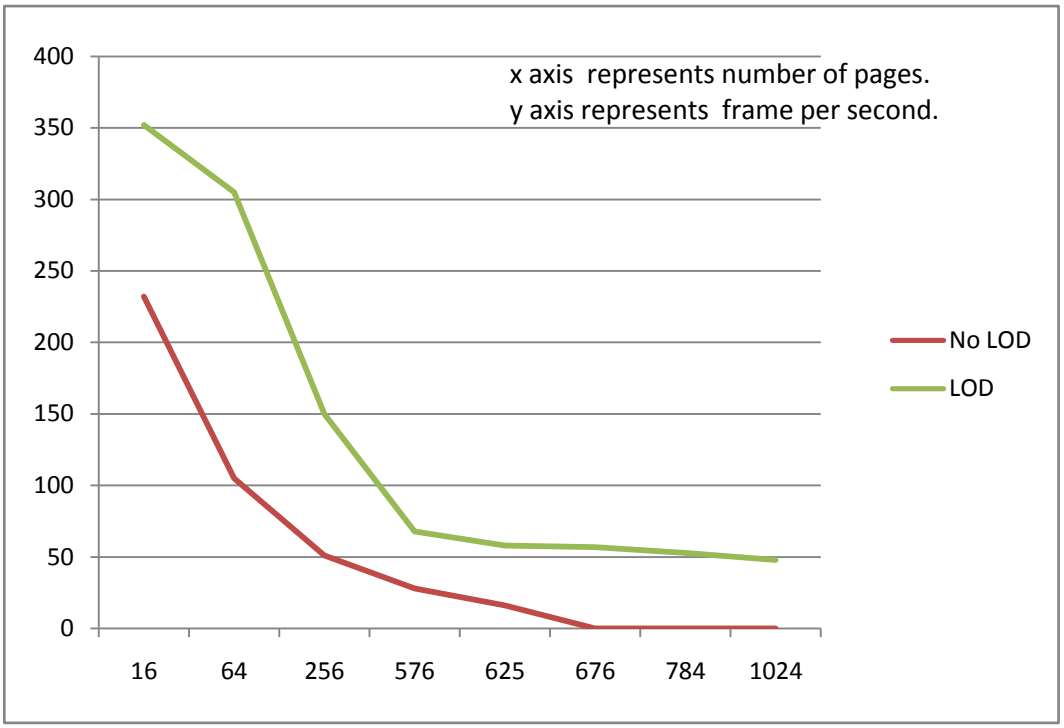

Figure 79 The performance graph with LOD and without LOD algorithms. The performance of algorithms which use LOD are higher than the algorithms without LOD.

5.2.2 Parallel Rendering

Recently, parallel rendering became an important research area by demanding high resolution graphics on simulation projects. To supply this demand, a parallel rendering engine is created.

As in mentioned in chapter one, there are several parallel rendering APIs. After a long research period and experiments, it is decided to use Equalizer as parallel rendering API. Equalizer does not fit our system directly, but the architecture of Equalizer is dynamic for modifications. So it is modified for our system.

Our system is an *OpenSceneGraph* (OSG) based rendering engine. Pure Equalizer does not include features of scene graph. It mostly works with pure OpenGL so the management of the scene is so complex.

The OpenSceneGraph is an open source, cross-platform graphics toolkit in order to develop graphics applications. It is based on “Scene Graph" architecture, providing an object oriented framework over the OpenGL. This makes it easier to implement and optimize low-level graphics calls [26].

The key strengths of *OpenSceneGraph* are its performance, scalability, portability and the productivity. [26]

OpenSceneGraph has many performance features, such as view-frustum culling, occlusion culling, small feature culling, Level of Detail nodes, OpenGL state sorting, vertex arrays, vertex buffer objects, OpenGL Shader Language and display lists as part of the core scene graph.

In order to achieve these features, an engine created which has a name Sim3D. This engine is an OSG based toolkit and it is developed in Simsoft Bilgisayar Teknolojileri Ltd. Şti.. Sim3D has many libraries, such as animation, audio, environment, terrain, road, water, effect, vegetation which enable developers to create a scene easily. Most of the implementations of this study which is about terrain rendering and editing are integrated with terrain library of Sim3D.

After integrating with the Equalizer, parallel rendering feature is attained. The application structure that works with the architecture of Equalizer is created. The Equalizer integration brought the feature of parallel rendering to Sim3D. Scene can be rendered in multi-pipe or multi-core systems. On this integration process, some major problems are handled. Such as:

1. Camera orientation problems on multi display scenes. Every camera displays the same place with the same point of view. A camera rotating mechanism is created dependent on a configuration file.
2. Keyboard and mouse input problems occurred on multi-screen systems. Since the main movement or rotation is done on the server side, inputs on the

rendering machines do not work. To handle these problems, a messaging infrastructure is created to send inputs from rendering clients.

CHAPTER 6

DISCUSSION AND CONCLUSION

6.1 Achievements

In this study, there are three important achievements. They are:

- Terrain Editing and Painting
- Terrain Paging and LOD Mechanism
- Multi-Screen Display

Terrain editing and painting can be performed easily and fast. Several algorithms are implemented and tested and the most suitable methods are chosen to this case. In editing, the VBO is used in order to change the position of vertices and the values of normals. In order to paint terrain, a combination of textures and shaders are used. This method brings fast and easy way of texture painting with very little memory consumption. After adding a paging mechanism to this editor, the capability of the terrain editor is increased. Since the terrain pages are loaded when needed and the old ones are unloaded on editing, the editor can create and manipulate infinite number of pages at interactive frame rates.

The terrain editor is integrated with GIS. This terrain editor tool can load real terrain data by the GIS integration. This feature enables to create height maps by selecting the area on the map. On editing, all the terrain manipulations and the object placements on the terrains are reflected to GIS data.

Terrain paging and LOD mechanism are implemented for very large terrains. Terrain paging is both implemented for the terrain editor and the terrain viewer. In editor, the system enables editing on multiple pages at the same time without noticed page borders. A LOD mechanism is created for pages in the terrain

viewer. Each page is loaded in different resolution dependent on the distance to the camera. Since loading process is done in threads, the performance of the system does not affect on loading. The LOD mechanism is not only for the geometries, but also for textures. The pages with less detailed geometries use low resolution textures that the difference cannot be noticeable by a human eye. The major performance improvement is achieved through LOD mechanism. This terrain viewer enables to walk around hundreds of kilometers on a terrain with interactive frame rates. This tool just needs to have the data of that large terrain. This viewer can be used with either a land vehicle or a plane. Since a land vehicle moves over the terrain, the resolution of the terrain should be high. The land vehicle moves slowly enough that the terrain loader can load new pages before this vehicle arriving next page. A plane moves much faster than a land vehicle. Normally, it is so fast that a terrain loader cannot catch to load new pages. The advantage of a plane is that it moves much higher than a vehicle. This situation brings the advantage of using low resolution pages on the most of the regions of terrains. Since a plane does not see the terrain in high resolution, terrain loader can load all low resolution pages without recognized by a human eye. This system can be used in an image generator system easily.

A major topic of this study is Multi-Screen Display Viewing and Parallel Rendering of a terrain. With the integration of the Equalizer and OpenSceneGraph, a system which enables multiple numbers of displays which have different point of views is created. This enables to display scene in environments like cave or dome systems. These systems are designed in order to increase the effect of realistic feel by projecting the scene to the walls of the room of dome or cave. The user of the system sees the scene wherever he looks in the room. This achievement can also be used in order to increase the resolution of the screen. Since the infrastructure enables multiple rendering nodes, the screen resolution is just dependent on the number of computers and monitors connected to these computers. There is no terrain viewer which has multiple screens like this application. After editing the terrain, the application displays whole terrain in

multiple screens with very high resolution and interactive frame rates. The number of GPU in display system increases the performance of the system.

6.2 The limitations of Current Work

There are some limitations which are not handled by this study. The first one is the limited number of pages rendered at the same time. Even if the paged LOD mechanism lets the tool load an infinite number of pages, rendering system have problems when the number of rendered pages is larger than 1024. The frame rate decreases dramatically when the number of pages increased too much. It is an undesirable result of an interactive system.

The second limitation is camera speed. When the camera moves so fast that the loading mechanism cannot load fast enough, pages are not displayed or they are displayed with very low resolution.

The third limitation is the area of the terrain. On measuring the total size of the terrain, the distance between points in the height fields are taken 1m. If the resolution of a terrain is desired to be increased to 0.5m or 0.25m, the total size of the terrain will be shrinking with proportion to the new resolution.

6.3 Future Work

In this study, some future works are also emerged. The first one is about handling pages whose has different resolutions. On terrain editing, the resolution of all pages are the same. Instead of having a constant resolution, there should be pages with different resolutions. These resolutions should increase by the demand of detail on that page. This improves the performance both on editing and on rendering. It also supplies an effective usage of memory.

The second one is about the cracks between pages. If two neighbor pages have different resolutions, then the cracks may occur between these two pages. When a high detailed page represents the same line with 256 values, a low resolution page

may represent the same line with 128 or 64 values. If the values on the high resolution page do not match with the interpolated values of low resolution page, cracks occur on that not matching area. Sometimes these cracks could not be very small and they could be noticeable by a human eye. In order to avoid these cracks, curtains are added to terrain borders. This method makes these cracks not noticeable but it is not an efficient way. A crack avoidance mechanism should be implemented that removes gaps between pages.

The third one is about LOD on editing. A LOD mechanism could be added on terrain editing and user can see whole terrain while editing. Different from the terrain viewer, LOD of pages should be changed by the active terrain instead of camera position. This improves the performance of the terrain editor.

Another future work is about multi-resolution on terrain deformation. Terrain resolution can be incremented while rendering if a deformation request comes. This deformation is mostly needed when a vehicle moves on a snowy terrain or an explosion occurs. These effects are expected to change the shape of terrain. The resolution of terrain could be changed dynamically and could be edited interactively. This feature could be useful for the simulation systems.

The last one is about the number of threads on loading pages with LOD. Since this process is an IO operation there is only one thread in order to load terrains. It may be dangerous to increment the number of threads since there is only one disk reader. If two threads try to access the same data on disk, there would be an exception. In order to handle this problem, there should be many threads that build terrains but one thread to read data from disk. This improvement would probably increase the terrain loading performance.

REFERENCES

[1] IEEE Transactions on Visualization and Computer Graphics, vol. 15, no. 3, pp. 436-452, "*Equalizer: A Scalable Parallel Rendering Framework*" IEEE, 2009.

[2] Equalizer web site, "*Chromium vs. Equalizer",* http://www.equalizergraphics.com/documents/user/crComparison.html, last accessed 27.06.2009.

[3] Equalizer Technical Report IFI-2007.06, *Department of Informatics, Unviersity of Z¨urich, 2007*

[4] Chromium web site, "*Chromium",* http://chromium.sourceforge.net, last accessed 27.06.2009.

[5] Humphreys G, Housten M, Ng R, Frank R, Ahern S, Kirchner P, Klosowski J, "*Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters", SIGGRAPH 2002*.

[6] Eilemann S, "*Equalizer Programming Guide", SIGGRAPH 2008, Article 45*

[7] Losasso F, Hoppe H, "*Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids", ACM Transactions on Graphics, ACM Press (Proceedings of SIGGRAPH 2004), Volume 23, 2004-10-13*

[8] Levenberg, J. "*Fast view-dependent level-of-detail rendering using cached geometry*.", In Proceedings of the 13th IEEE Visualization 2002 Conference, Boston, 2002.

[9] Sümengen S, Balcısoy S, "*Hava ve Kara Araç Gruplarının Detaylı Arazi Verisi Üzerinde Gerçek Zamanlı Simülasyonu*.", *HITEK, Istanbul, 2004*

[10] Asirvatham A, Hoppe H, *"Terrain Rendering Using GPU-Based Geometry Clipmaps", GPU Gems 2,* NVIDIA

[11] Yin P, Jiang X, Shi J, Zhou R, *"Multi-screen Tiled Displayed, Parallel Rendering System for a Large Terrain Dataset",* The International Journal of Virtual Reality, 2006, 5(4):47-54

[12] ScalableGraphics web site, "*Chromium",* http://www.scalablegraphics.com/index.php?l1=products&l2=dtc2, last accessed 27.06.2009.

[13] OpenGL Multipipe™ SDK , *"OpenGL Multipipe™ SDK White Paper"* , 007-4516-002

[14] OpenGL Multipipe™ SDK web site, *"SGI – Products: OpenGL Multipipe SDK Home Page"*, http://www.sgi.com/products/software/multipipe/sdk/, last accessed 04.07.2009.

[15] FreeWorld3D web site, *"FreeWorld3D 2.0 – Terrain Editor and World Editor"*, http://freeworld3d.org/features.html, last accessed 06.07.2009.

[16] Marselas, H. "*Optimizing Vertex Submission for OpenGL"*, Game Programming Gems, pp. 353-360.

[17] Szoka E. "*Triangle Strip preserving LOD (T-Strip LOD)"*, Computer Science 95.495B Honours Project, Carleton University, April 2002

[18] EarthSculptor web site, *"EarthSculptor - Technology"*, http://www.earthsculptor.com/technology.htm, last accessed 06.07.2009.

[19] PnP TerrainCreator web site, *"PnP Terrain Creator - Home"*, http://www.pnp-terraincreator.com/, last accessed 06.07.2009.

[20] Artifex Terra 3D web site, *"Artifex Terra 3D Terrain Editor – Easy and sophisticated landscape painting and editing"*, http://www.artifexterra3d.com, last accessed 06.07.2009.

[21] G. Humphreys and M. Eldridge. "*WireGL: A Scalable Graphics System for Clusters, In: Computer Graphics Proceedings"*, Annual Conference Series, ACM SIGGRAPH, pp. 129~140, Los Angeles, California, 2001.

[22] Using Vertex Buffer Objects - White Paper, NVIDIA

[23] G. Humphreys and M. Eldridge. "*Distributed Rendering for Scalable Displays"*, *Proceedings of SC2000, Article no. 30.*

[24] P. Heckberd and M.Garland. "*Survey of Polygonal Surface Simplification Algorithms"*, *Multiresolution Surface Modeling Course, SIGGRAPH '97*

[25] Wolfire Blog site, *"Wolfire Games Blog"*, http://blog.wolfire.com/, last accessed 15.08.2009.

[26] OpenSceneGraph website, *"About/Introduction - osg"*, http://www.openscenegraph.org/projects/osg/wiki/About/Introduction/, last accessed 18.08.2009.