COLLAPSIBLE
3D GIS VISUALIZATION

by

Suwen Wang

Submitted in partial fulfillment of the
requirements for the degree of
Bachelor of Computer Science, Honours

at

Dalhousie University
Halifax, Nova Scotia
March 2007

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Computer Science for acceptance a thesis entitled "COLLAPSIBLE 3D GIS VISUALIZATION" by Suwen Wang in partial fulfillment of the requirements for the degree of Bachelor of Computer Science, Honours.

Dated: March 30, 2007

Supervisor: _____
Professor Stephen Brooks

Reader: _____
Professor Dirk Arnold

DALHOUSIE UNIVERSITY

DATE: March 30, 2007

AUTHOR: Suwen Wang

TITLE: COLLAPSIBLE 3D GIS VISUALIZATION

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: B.C.Sc. (Honours) CONVOCATION: May YEAR: 2007

Signature of Author

Table of Contents

List of Figures

Abstract

Visualization has always been an essential task in the application of geographic information systems (GIS). Increased computational power has made 3D GIS visualization widely available. With it came the greater need for interactivity and intuitive control through user interfaces.

My thesis describes the design and implementation of a 3D GIS visualization application featuring "terrain collapsing", a new user control functionality with which part of the terrain can be folded. It is useful for visually comparing two distant terrain patches. Real-time adaptive quadtrees are used as LOD structure for the terrain mesh, and efforts to optimize rendering are discussed.

Acknowledgements

I sincerely appreciate the help and guidance Dr. Stephen Brooks gave me, and the advice from members of our Computer Graphics and Visualization Group, especially the helpful input from Rafael Falcon lins. Discussing and communicating ideas with them has been an inspiring experience. I also want to thank my parents, aunt and uncle for their care and support.

Chapter 1

Introduction

1.1 Motivations

Because of the spatial nature of the data stored in a geographic information system (GIS), visualization has always been an essential component of GIS functionality. Proper visualization facilitates better understanding of the data and enables GIS users to make intuitive estimation and inference.

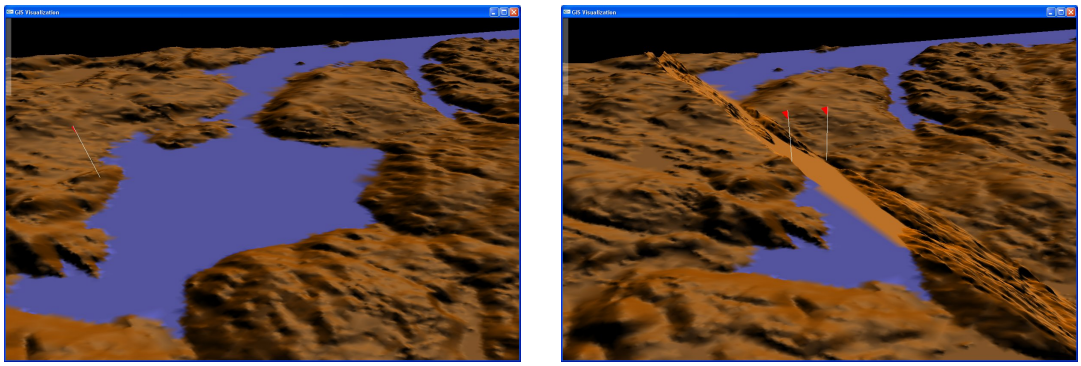
With the stronger graphics capabilities in modern workstations, 3D GIS visualization has become increasingly accessible. Visualizing geo-referenced data in 3D virtual space can enhance the presentation of their spatial relations, and the additional dimension over 2D visualization increases the flexibility of data representations.

However, there are also challenges for 3D GIS visualization. With more flexibility to present the information comes a greater need for good interactivity between the system and its user, which requires the graphics rendering executes with acceptable speed. Therefore one important goal for a 3D GIS visualization is to optimize the performance of the rendering.

1.2 Goal

The goal of our research is to build a GIS visualization application. Given the corresponding topographic data, a middle-scale terrain will be modeled in 3D virtual space. GIS data can be visualized by overlaying on the base terrain model. Besides common user interface controls for navigating in the 3D space, our application will provide the user with a new control functionality to fold or collapse a portion of the terrain and bring the unfolded terrain regions together so that they can be viewed in

the same screen, as shown in Figure 1.1. The scope and orientation of the collapse is determined by user input. This functionality may be useful for visually comparing two distant terrain patches.



(a) (b)

Figure 1.1: Terrain Collapsing: Bedford Basin

1.3 Outline

Chapter 2 is a brief survey of the previous research related to 3D GIS visualization, which are divided according to their main emphasis. In Chapter 3, we describe the technical choices and techniques used in basic system setup. Chapter 4 details the implementation of our terrain mesh structure, including the rationale of implementation choices and the obstacles we overcame. Our efforts for rendering optimization is documented in Chapter 5. Chapter 6 describes how we implemented the terrain collapsing interface. Chapter 7 briefly discusses the experiments we did and presents the result and analysis. Finally, we conclude in Chapter 8 and suggest possible future work.

Chapter 2

Related Work on 3D GIS Visualization

As most of the data in a geographic information system is spatially referenced, visualization is an essential step to help people to understand and utilize these data. Properly visualized data can convey the spatial relationship between the depicted geographic features, and facilitate quick learning and intuitive inference by the user.

In the early stage of GIS visualization development, 2D technologies were first applied to visualize geographic data. Research done on digital cartography have been enabling scientific and industrial communities to carry out various spatial surveys and analysis with digital 2D maps. Like many digital information formats, the digital counterpart of the traditional map has the distinct advantages of fast querying and easy updating.

Until relatively recently, the main factor that hindered the development and deployment of 3D GIS visualization has been high demands of the computing power and graphical capability. With the significant improvement of the hardware capabilities in recent years, 3D GIS visualization has become common practice, and recent research has focused on this active area. In the rest of this chapter, I will briefly review the relevant research in terms of topology modeling, user interface design, and distortion techniques in visualization.

2.1 Topography Modeling

Most terrain rendering systems receive input data for the terrain topography in the Digital Elevation Model (DEM) format, which stores the ground elevation measures in a regular-spaced grid. Therefore, the most straightforward way to model the terrain surface is to construct a regular grid. However, as pointed out by Silva et. al., it was

beyond the capability of real-time systems to fully render DEM when the study was done because of the large number of vertices required to model the DEM [1]. Today, although DEM can be rendered using real-time systems, it includes more detailed geometry than necessary and interactive graphics performance is difficult to achieve.

Among the alternative mesh models proposed to solve this problem, an irregular triangle network (TIN) is commonly used. Several efficient methods of generating TINs from DEMs have been proposed in [2], [3], and [4]. However, the simplifications for constructing TIN do not preserve the regularity of the grid, which is desirable for applying certain types of level of detail (LOD) techniques [5]. As another school of meshing structures, quadtrees keep the geometric regularity so that different levels of geometric detail can be generated. The quadtree data structure has been well studied and documented, for example, in [6] and [7]. Various techniques to construct quadtree from height-field data set have been mentioned by [8], [9], and others.

2.2 User Interface Design

In terms of user interface design, relatively little research has been done for 3D GIS visualization when compared with 2D visualization. The overwhelming impression about GIS software usability used to be that the users need strong geo-science domain knowledge in order to understand or even navigate in the GIS software [10]. Since then, the GIS systems have become more user-friendly. Other developments have concentrated on specific application scenarios and the effectiveness of the GIS system. For example, Pyush Agrawal and his colleagues designed a GIS user interface system used for crisis management [11]. The system recognized natural human gestures so that the efficiency of the system is maximized. A novel visual-query-by-sketch interface has been proposed in the attempt to make the GIS systems more accessible for non-expert users [12].

However, the future looks more optimistic for 3D GIS interfaces. In a usability study for a real-time 3D GIS system, US army commanders could successfully assign tasks according to the virtual terrain, and they concluded that the immersing capability may improve the navigational accuracy [13]. Finally, a layered GIS content

display interface has been developed to enhance the visualization capacity [14].

2.3 Distortion Techniques in Visualization

Large scale information systems usually suffer from the problem that the screen space is only sufficient for visualizing localized data set, providing little clue about how these data relate to the rest of the system. Distortion techniques have been adapted to overcome this, providing a detailed focus on local data set and peripheral global context at the same time [15]. Distortion effects in graphical data representation are generally achieved by applying transformation to individual data objects. In the study of the distortion techniques, the amount of distortion is usually measured by a class of functions called magnification functions.

Various techniques have been proposed to achieve distortion-oriented visualization. Leung and Apperley classified these techniques into two types according to their magnification functions [15]. One type of distortion techniques is characterized by a piecewise continuous magnification function, where there are abrupt magnification changes across the graphical representation. Classical examples of this type of technique are bifocal display [16] and perspective wall technique [17]. The other type of distortion display has a continuous magnification function. Polyfocal display [18] and fisheye projection [19] both belong to this type. While all the techniques mentioned above are for 2D visualization, M. Sheelagh T. Carpendale, David J. Cowperthwaite, and F. David Fracchia explored the possible application of distortion in 3D visualization [20].

Chapter 3

Basic System

For our 3D GIS application, we decided to implement it with the OpenGL API. As a well established graphics programming library, OpenGL provides graphics developers with powerful capabilities and extensive functionalities. But with its wide range of technical possibilities, care must be taken to properly setup the application so that it can achieve the desirable effects. Therefore, to setup our application, there were several technical issues we needed to solve in order to use the OpenGL graphics abilities and achieve expected system behavior. In this chapter, we will discuss the design choices and rationales for each implementation choice.

3.1 Base Terrain Modeling

3.1.1 Naive Approach for Grid Modeling

As the foundation of our GIS visualization, the terrain model is very important. Polygon meshes are often used to model the terrain surface. After the grid points are specified by the elevation data, the corresponding vertices are usually grouped to define geometric primitive objects, which in turn forms the entire terrain mesh.

As we mentioned in the previous chapter, the DEM files provide the elevation data for each of the vertices on a regular grid, which covers certain land area. As a starting point, we decided to use quadrilateral mesh to visualize the regular data grid on which the vertices were define. To store the data, a regular 2-dimensional array is used, so that a 1-to-1 mapping is formed between the array indices and the locations of the vertices in $x - z$ plane. For example, if the size of the row is n, to obtain the coordinate for the j^{th} point on the i^{th} row, the program will access the $(i \times n + j)^{th}$

element in the array.

Once the elevation data grid is in place, we need to decide which geometric primitives these vertices will form. Geometric primitives in OpenGL are defined by first identifying its object type and then defining each of its vertices in an appropriate order. The simplest approach to build geometric primitives using our grid data is to define each cell in the grid as an individual quadrilateral. The program can iterate over the whole grid and define a separate quadrilateral using four vertices surrounding a single cell.

However, this approach is too crude to be practical. In OpenGL API, the same geometry can usually be defined by different combinations of primitive object definitions. Because the built-in OpenGL functions are normally optimized to utilize the capability of the graphics hardware, they are generally much more efficient than the computation of lower level functions with the same overall effects. Also, transfering data from the system memory to the on-board memory of the graphics card is the bottleneck of many graphical rendering processes. The functions defining more complex geometry can usually do that without the application specifying every single vertex, so that the total amount of data to be transfered to the graphics card is reduced. So it is desirable to reduce the number of function calls and use complex functions to generate geometry.

Therefore, the previous naive approach of grid generation is a brute force attack to our problem. The large number of function calls to define individual quadrilaterals can be replaced by fewer function calls, each of which covers larger portion of the terrain geometry. Moreover, each vertex registration in the geometric definition is also a function call. With the naive approach, the two vertices on a shared grid edge are defined twice when the two adjacent cells are defined. This almost quadruples the total number of function calls to define vertices.

3.1.2 Improved Mesh Generation

We improved the mesh generation by defining each row of the grid as an OpenGL quad-strip object, so only one object definition function call is needed for the entire row. Furthermore, a quad-strip object is defined by sequentially defining the vertex pairs extending the strip, so the number of the redundant vertex definition calls are reduced.

Using quad strips to form the grid also provides a generic method to divide the continuous surface mesh into discrete geometric "building blocks". For the collapsing functionality, the whole terrain needs to be partitioned into sections, so that different methods can be applied to different sections to achieve the collapsing effects.

3.2 Lighting and Shading Setup

3.2.1 Basic Concepts

In 3D graphics, lighting and shading are crucial elements for the perception of dimensionality. Without the effects of shading, objects in the virtual scene do not have shadows, making it impossible to distinguish the different facets of an object. Because the interaction between light and objects in the real world is very complex, OpenGL uses the Phong shading model to simulate lighting effects. In the Phong shading model, the shade on a surface is determined by the properties of the light source, the surface material, and the surface orientation with respect to the light source and the location of the viewer. In our application, only two sets of surface material properties were defined to distinguish land mass and water body, so any point on the land region has the same surface material properties. A constant white parallel light source was used to illuminate the terrain. Therefore, the different orientations of the points on the terrain surface is the only factor to determine different shades and achieve 3D realism.

To determine the orientation of a polygon surface, OpenGL requires the application to provide a normal vector for each vertex on this surface. By definition, given

a surface, the normal is a vector that is perpendicular to the surface, so it is perpendicular to any vector that is in the surface. Therefore we can compute the normal of a surface by taking the cross product of any two non-parallel vectors on the surface. The following is the detailed calculation, and the concept is depicted in Figure 3.1:

Let $\overrightarrow{AB}$ and $\overrightarrow{AC}$ be two vectors in a plane; and

$$\overrightarrow{AB} = a_1\overrightarrow{i} + a_2\overrightarrow{j} + a_3\overrightarrow{k};$$
$$\overrightarrow{AC} = b_1\overrightarrow{i} + b_2\overrightarrow{j} + b_3\overrightarrow{k};$$

Then the normal of plane ABC is :

$$n = (a_2b_3 - a_3b_2)\overrightarrow{i} - (a_1b_3 - a_3b_1)\overrightarrow{j} + (a_1b_2 - a_2b_1)\overrightarrow{k} \tag{3.1}$$

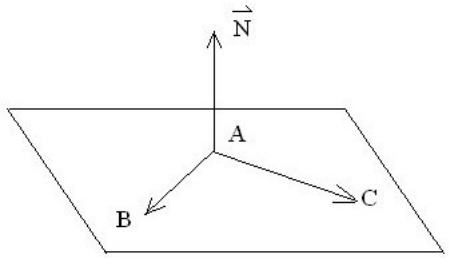


Figure 3.1: Calculate the Surface Normal

Note that, specifically for a triangle, the cross product of any pair of its edges will be the normal for the plane which the triangle is in. We will use this method to compute vertex normal.

3.2.2 Averaging Vertex Normals

To accurately approximate the surface normals on the individual vertices in our DEM data grid, we computed normal value for each vertex, regardless of which of them will be used to form the final terrain mesh. In the regular grid formed by DEM data

points, each vertex is shared by 4 or, in the boundary cases, 2 surrounding cells. The question we are facing is how to determine the normal on this vertex. To better model a smooth surface, we computed the normal values for the surrounding cell surfaces, and then take the average as the normal on the central vertex. In each grid cell, the two sides of the quadrilateral intersecting at the vertex can readily serve as the two surface vectors for normal computation. The Figure 3.2 shows how the normal on a vertex is calculated from surrounding cells.

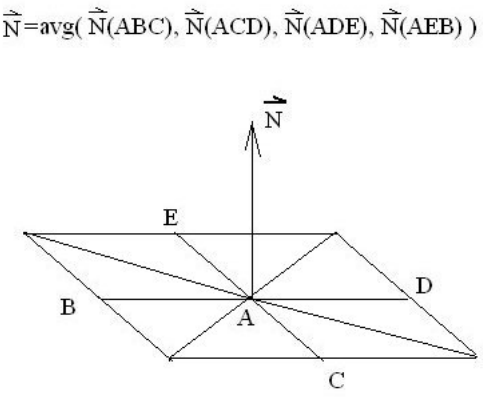


Figure 3.2: Calculate Averaged Vertex Normal

But we quickly realized this normal computation is not good enough, and the shading on testing render image was poor. After closer investigation, we realized the problem was that a single normal is not sufficient to represent the orientation for a grid cell, because the four vertices forming the cell are very likely not in a single plane, except in the regions where the terrain itself is flat, for example, on the water surface.

So we increased the accuracy of the vertex normal averaging by computing two normals in each surrounding cell, so that the normal for a vertex surrounded by four cells is averaged from eight triangle surface normals around it. With the refined average normal, the shading quality improved significantly when the terran is viewed locally.

3.2.3 Normal Caching

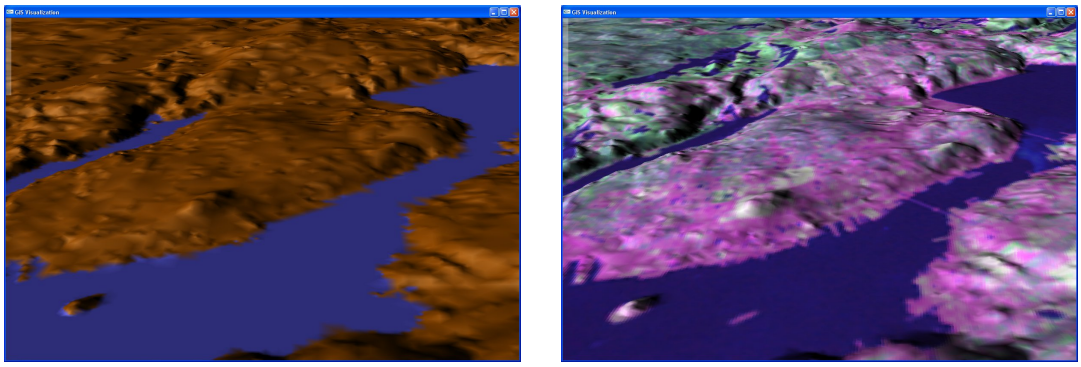
When we first implemented the normal calculation, the terrain display became extremely slow. After a quick check, we found out the problem was that the normal calculation routine had been invoked within the display routine. There are two serious problems with this. First, because the geometry of the base DEM grid never changes, the localized normal values for each vertex is constant. So recomputing normals in each frame is totally unnecessary. Another problem with this is that even if some vertices will not be used for the current terrain mesh, their normals would still be recomputed and reset.

We recognized that the normal computation is clearly a preprocessing step and can be carried out once as the system initially setup. Also, because the geometry of the terrain is constant, we stored the vertex normal as an extra attribute for each vertex. Therefore, after the initialization, whenever a vertex needs to be rendered, we directly looked up its normal value without any extra computation.

3.3 Texturing with GIS Thematic Map

We visualize the thematic GIS data by mapping the data image on to the terrain model as a texture, as shown in Figure 3.3. Our system uses the GIS thematic data in the form of map images. These images were the 2D representations of the thematic data. To simplify the project, we first converted all the images into the simple PPM format. This simple format allows us to directly read in the RGB color values from the file. The image data read in from the image file are then stored. To use it later as a texture object, the storage array used must be a one dimensional array, because the function to specify texture object can only process one dimensional arrays.

One important thing to note is that we need to control the pixel store states to ensure the reading and writing of pixel values work as we expected. In particular, OpenGL set the default value of pixel alignment to 4, which means there are 4 bytes padding around the data block storing the pixels. In order to have desirable system behavior, we change the setting so that the pixel data is stored without paddings.



(a) (b)

Figure 3.3: Normal Rendering and Texture Mapping Rendering

3.4 Interactive User Navigation

Using keyboard and mouse as input devices, the user can interactively control the visualization and navigate the virtual camera above the terrain. The keyboard input is used to control the environment setup, such as whether the texture mapping is applied, and the camera navigation. Different key sets are used to simulate the pitch, yaw, and forward and backward movement of the camera. The roll movement is not included, because it is not suitable for our visualization purposes.

We defined two modes in which the mouse can be used. By default, the user can use mouse to specify locations on the terrain surface, where the movement of the mouse does not affect the navigation. We call this "selection mode". The selection functionality will be further discussed in the next section. By clicking the middle mouse button, the user can switch between the selection mode and the navigation mode. In navigation mode, the motion of the mouse is used to determine the forward,backward and yaw movements of the camera.

3.5 Point Selection on Terrain Surface

3.5.1 Mouse Picking

In order to let the user interact with the terrain, we need to provide the user with the ability to specify a location on the virtual terrain with a click of the mouse. We want the mouse picking functionality to be intuitive, so that the user can click on a point on the rendered 3D terrain surface and get exactly that point selected. It was left for the application program to recognize where on the $x - z$ plane this selected point really was and use the resulting coordinates to specify the reaction to this user input.

However, there were several challenges for this approach. First, when the user clicks on a point in the viewport, the action does not specify a single point in the 3D space because of the depth in the 3D scene. Instead, the mouse click casts a imaginary ray extending from the point of view to infinity. For our application, we must determine which point on this ray is the intersection point with the terrain surface, which is the exact location the user intended to pick. Secondly, even after we fixed the location of the picking point, it will still be expressed in eye coordinates; and the camera is almost certainly not in a standard position, so in order to obtain the object coordinates for the point of interest, we also need to perform transformations between the object and eye coordinate systems. Yet another challenge is that the terrain surface has irregular and complex geometry. It is difficult to find the exact location of the intersection point regarding to the local terrain geometry.

3.5.2 Intersecting Ray with Base Plane

To explore the possible solutions to this problem, our first attempt is to find an approximation of the point location, which is easier to calculate. For this, we investigated a similar but more straightforward question, which is how to determine the coordinates of the intersection point of the mouse picking ray and the horizontal plane at $y = 0$.

From a mouse click, the OpenGL standard mouse function returns the screen coordinates (u, v), in screen coordinates. In our application, we keep track of the current camera position and orientation. The camera position is recorded as coordinates (x_C, y_C, z_C), and the orientation is encoded and stored in two variables θ_x and θ_y, which denote the accumulative angles the camera rotated along x and y axis from the initial direction, respectively. We chose plane $y = 0$ as our target horizontal plane. Therefore, this simplified problem can be solved as following:

Given the click location (u, v), where v is converted to start from the bottom of the screen, a parameterized line can be formulated with respect to the camera coordinate system:

$$\begin{cases} x_v &= \frac{2u-W}{W} z \tan(\frac{\text{fov}}{2})(\frac{W}{H}) \\ y_v &= \frac{2v-H}{H} z \tan(\frac{\text{fov}}{2}) \\ z_v &= -z \end{cases} \qquad (1)$$

where W, H are the width and height of the vertical clipping plane of the viewing volume, and fov is the field-of-view angle; they are variables set by gluPerspective(). Note that z is always positive, because it is the distance of the points on the cast ray from the eye point. If we can find a parameterized equation for the ray line with respect to the object coordinate system:

$$\begin{cases} x_w = f_x(z) \\ y_w = f_y(z) \\ z_w = f_z(z) \end{cases},$$

where z is the line parameter in the first line equation; then we can solve $y_w = f_y(z)$, in our case y_w is 0. After getting the value of z, we can solve for x_w and z_w, hence the object coordinates of the intersection point.

This becomes a change of coordinates problem. Let P_v be a point in camera coordinate system, and P_w be the same point represented in world coordinate system.

$$
\begin{aligned}
P_w &= \mathbf{T}(x_C, y_C, z_C)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)\mathrm{P}_v \\
&= \begin{bmatrix} 1 & 0 & 0 & x_C \\ 0 & 1 & 0 & y_C \\ 0 & 0 & 1 & z_C \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y & x_C \\ 0 & 1 & 0 & y_C \\ -\sin\theta_y & 0 & \cos\theta_y & z_C \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} \cos\theta_y & \sin\theta_x\sin\theta_y & \cos\theta_x\sin\theta_y & x_C \\ 0 & \cos\theta_x & -\sin\theta_x & y_C \\ -\sin\theta_y & \sin\theta_x\cos\theta_y & \cos\theta_x\cos\theta_y & z_C \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} x_v\cos\theta_y + y_v\sin\theta_x\sin\theta_y + z_v\cos\theta_x\sin\theta_y + x_C \\ y_v\cos\theta_x - z_v\sin\theta_x + y_C \\ -x_v\sin\theta_y + y_v\sin\theta_x\cos\theta_y + z_v\cos\theta_x\cos\theta_y + z_C \\ 1 \end{bmatrix}
\end{aligned}
$$

$\therefore$ Use x_C, y_C, z_C from (1), and solve with $y_w = 0$:

$$
\begin{aligned}
\because \quad y_v\cos\theta_x - z_v\sin\theta_x + y_C &= 0 \\
\tfrac{2v-H}{H} z\tan\left(\tfrac{\text{fov}}{2}\right)\cos\theta_x + z\sin\theta_x + y_C &= 0 \\
\therefore \quad z = \tfrac{-y_C}{\sin\theta_x + \frac{2v-H}{H}\tan\left(\frac{\text{fov}}{2}\right)\cos\theta_x}&.
\end{aligned}
$$

Using the formulas we derived, the program can directly compute where on the $y = 0$ plane the mouse clicked. We implemented this functionality, and the program correctly identifies a mouse picking location on the $y = 0$ plane.

However, when we tried this method with the actual terrain model, the results of identifying click points were far from satisfactory. After some further consideration,

we concluded the assumption we made about terrain surface elevation was not appropriate. The distance from the target point on the terrain surface to the base $y = 0$ plane is not negligible compared to the size of a grid cell. So the point determined by the above calculation is almost always not accurate enough to approximate the target. Even worse were the cases where the mouse clicks on a hill slope facing the camera. Because of the increased elevation, the intersection point between the mouse clicking ray and the base plane was much further away from the accurate vertical projection of the point of interest. Clearly, we needed some other techniques to solve this problem.

3.5.3 OpenGL Selection Mode

After a broader survey of the background knowledge about OpenGL, we found that there is architecture called selection mode to handle the mouse picking problem. We will briefly discuss the characteristics of the selection mode, and discuss whether it is suitable for our application.

The OpenGL rendering mode controls how the virtual scene is rendered. There are two rendering modes: normal rendering mode and selection mode. For most of the 2D or 3D rendering, the normal rendering mode is used. In this mode, the virtual camera is transformed to a desirable vantage point, and the scene is clipped against a viewing volume in front of the camera. Only the portion of the scene included in the viewing volume is deemed visible, and rasterized and sent to frame buffer. Finally the content of the frame buffer is rendered by the hardware.

The other type of rendering mode, the selection mode, is designed to handle mouse picking. Here all the viewing calculations are similar to those in normal mode, but the differences are the shape of the viewing volume and the rendering destination. In the selection mode, the viewing volume is a small region of space centered at the ray cast by the mouse click. The content within this minute viewing volume, if any, is sent to a special memory buffer instead of the frame buffer, thus no image will be displayed. For each predefined model object that is either partially or completely rendered in this miniature viewing volume, a signal called “hit” is generated and the label for

this object is recorded into a list. Therefore, if the objects are carefully labeled, the clicked object can be found among a few objects in the hit list. If we know the depth value for these candidates, the picking target can then be determined.

However, after some further consideration, we decided it is not suitable for our application. Because the terrain model is a continuous mesh surface, no accurate method can be used to partition the whole mesh into discrete objects. If we define each grid vertex as an object, it is then not clear which of the surrounding cells are picked from a hit vertex. Also, this approach still requires that the depth of objects is known, therefore does not completely solve our problem.

3.5.4 Final Approach: Reverse Projection

The close relation between the point depth and its location is significant. If the depth of the point we want to identify on the terrain surface can be found, we can figure out the coordinates for the point using the formulas we described earlier. Then, we came across an OpenGL function glReadPixels(), which can query the properties of a block of pixels centered at the point of mouse click. Better yet, applying this function with a 11 pixel block, we can immediately get the depth value of the current pixel.

There are slight complications. The depth value returned from the glReadPixels() is a float number ranging from 0 to 1. For a selected point on a ray, it has the minimum depth value 0 when the point is the intersection between the ray and the near clipping plane, and the maximum value 1 when it is the point where the ray intersects with the far clipping plane. So there are some extra steps needed before we can get the actual depth value of the point.

We then discovered that OpenGL has built this reversed transformation process into one of its functions. The function gluUnproject() make the reversed transformation of the projection from 3D space to 2D viewport space, therefore encapsulating all the calculation we presented previously. And the format of the depth parameter is exactly the same as the return value from glReadPixels(). Thus, using these two powerful OpenGL functions, we can finally trace a mouse click from a point on the

screen to the target point on the terrain surface, and locating with mouse clicks was made possible.

Chapter 4

Implementation of Real-Time LOD Using Quadtree

4.1 Purpose of LOD

Most graphics and visualization packages require the ability to interact with users. How quickly the user input takes effect determines the fluency of the interactive experience. Most graphical applications achieve real-time responses to user input by constantly refreshing the graphical display. Once a user input is processed, the corresponding effects will take place the next time the graphical display is refreshed. The speed of screen refreshing is usually measured by how many frames the application can render within a second, where a frame is a complete image in the application window. A graphics application with higher refreshing rates takes less time for the user input to take effect. Therefore, it is crucial to ensure certain frame rate standards in order to create a seamless interactive experience.

However, the frame rate of a graphical application is restricted by the available hardware resources and the geometric complexity of the objects we want to render. A scene containing objects with high geometric complexity requires longer time to be rendered, resulting in lower frame rates. On the other hand, most objects in real life have infinitely complex shapes. The more complex the geometry of the virtual objects is, the better they can approximate their real life counterparts. When a 3D graphical scene can approximate the scene in real life well, we say it achieves a high fidelity; otherwise, we describe it as a low fidelity. So, like many other scenarios in computer science, given a fixed amount of computational resources, any graphical application faces a trade-off between frame rate and fidelity.

In our terrain rendering application, both the fidelity and frame rates are important [21]. As a GIS visualization system, displaying terrain realistically is an essential

goal. To achieve the required high fidelity, the mesh used to model the terrain surface needs to be as loyal to the elevation data as possible. For a surface mesh, the total number of the triangles represents its geometric complexity. Given an elevation data set with 1201 by 1201 data points in a rectangle grid formation, if every data point is used, the resulting mesh will have over 2.8 million triangles. Although only part of the entire mesh will be rendered most of the time, it still requires a large amount of computation to model every data point as an individual vertex. The visualization system should also accommodate the ability to interact with the user. Using keyboard or mouse, the user may adjust the position of the virtual camera over the terrain so that a simulated aerial views of the terrain surface can be obtained. To ensure a smooth interactive user experience, the frame rate for the terrain rendering should reach at least 10 frames per second [22].

4.2 Existing LOD Techniques: Classification, Strength and Weakness

To achieve better fidelity and frame rates so that a better overall performance can be achieved, a class of techniques called level-of-detail (LOD) is applied in most modern graphical applications. The principle of LOD is that some complex geometry in the rendering model can be substituted by simpler geometry if the substitution does not impact visual quality significantly. The LOD techniques were first purposed by James Clark in a 1976 paper, in which he recognized the inefficiency of rendering a cluster of geometry into several pixels [23]. With simplified geometry, the rendering time for each frame is reduced, hence a better frame rate is achieved while maintaining good fidelity.

There are three major types of LOD frameworks, which are discrete LOD, continuous LOD, and view-dependent LOD [24]. The discrete framework is the earliest approach, followed by the continuous run-time framework, which provides characteristics and strengths that the discrete framework does not have. The view-dependent approach is a more sophisticated extension based on the continuous framework, where the position and orientation of the virtual camera is taken into consideration when the proper levels of details in different regions of the model are determined.

In the discrete LOD framework, several versions of one object are built offline. Each of these versions has a different level of geometric details. At run-time, the program selects an appropriate version of the object to render. With the less-detailed version of the object, there is less geometry to render; therefore rendering time is reduced while the visual quality is maintained. Because the LOD object construction is done offline, only the selection of the proper object is needed, and no additional computation is required.

The continuous LOD framework differs from the discrete one in how the geometric details are stored. Instead of building several versions of the same object with different level of details, the continuous LOD stores the geometry in hierarchical data structures, where a spectrum from the coarsest geometry to the finest details is formed. The appropriate set of geometry is determined and rendered at run time. The most remarkable advantage of this approach is that it makes the gradual change of the details possible. With the minimum number of polygons rendered for desirable visual quality, higher fidelity can be achieved with a fixed polygon budget.

With the previous two types of LOD frameworks, an individual object has a single level of detail. However, this is not sufficient when the object we need to render has large dimensions. The object may stretch across several regions where different levels of details are appropriate. The view-dependent LOD framework can be applied to solve this problem. As an extension of the continuous LOD framework, the view-dependent LOD computes the geometry details for each frame, taking the position of the viewpoint into consideration. The levels of geometric details are determined based on their distances from the point of view, regardless which object they belong to. Therefore, the same object may have different detail levels on different parts of its surface; and the same part of the object surface may present different LOD in different viewing image. This framework provides extra flexibility such that objects can free up more polygon budget at regions of its surface where finer details are not needed.

In our project, the goal is to render a realistic 3D terrain model with interactive frame rates. Because the terrain surface is modeled by a large triangle mesh, the distances from the virtual camera to different patches of the terrain surface will

inevitably vary significantly. This leads to the ill representation of different terrain patches, such that the close-by patches occupy much more area than the ones with the same area but further away from the camera. Thus, the LOD techniques should be applied to make the rendering more efficient. Furthermore, as the terrain is a continuous surface, it requires different levels of geometric details for different parts of its surface and the changes between them should be gradual and less noticeable. Therefore, we chose to implement the view-dependent LOD framework using a real-time adaptive quadtree data structure.

In our project, the terrain mesh is generated from a height field. Some unique features of the height field make the quadtree data structure more suitable to our application. Terrain elevation data represented by height field has been described by researchers as "2.5-D", because each point in the two-dimensional base plane has only one corresponding elevation value [25]. Therefore, we can often simplify the processing and rendering from three dimensional operations down to two dimensional ones. Among the 2D spatial data structures, quadtrees have been well studied and documented. Quadtree is a hierarchical data structure that is frequently used for representing terrain meshes because of its high efficiency [26]. A quadtree representation of a surface is defined by recursively subdividing the surface to form the tree, in which each node associates with a portion of the surface. The area covered by each node is divided into four quadrants, which form the four children of this node, and the root covers the entire surface we want to model.

The advantage of this quadtree representation is that it provides a natural layout and effective controls for the LOD techniques. In the quadtree, each child provides finer geometric details for a quadrant of the parent block. Wherever more resolution is needed for a certain patch of the terrain, the program will traverse deeper to the children of the node representing the terrain region of interest. And when the geometry is detailed enough, the recursion can simply stop and extract and render the geometry encoded in the current node. With the certain flexibility provided by the quadtree data structure, the view-dependent variation of surface details can be readily implemented.

4.3 Implementation of A Real-Time Quadtree LOD

4.3.1 Naive Approach

As a starting point, we implemented a simplified quadtree structure and a naive rendering procedure. We first construct the quadtree from the regular grid height field data. Then we dynamically extract the geometry from the preprocessed quadtree according to some naive criteria and rendering them recursively.

Initially, the elevation data parsed from the DEM files are stored into a two dimensional array of vertex objects, where each object records all the related information about the vertex. The row and column of the 2D array represents the x and z directions in the 3D space, so that the elevation data stored has the same layout as the terrain model on the $x - z$ base plane. Thus, we can map the spatial locations of the data points to the indices in the 2D array, and manipulating vertices in 3D space is simplified to index operations. Then we recursively constructed the quadtree data structure by creating node objects that references the vertices that form the current quadtree block. Also, to form the tree, each quadtree node points to its four children which represent the four quadrants of the current block. The whole data set is recursively subdivided until a leaf node is reached where the vertices in the block are adjacent to each other in the elevation data set. Therefore, the leaves encode the finest geometric details provided by the DEM data.

When the quadtree structure is computed, the rendering process dynamically extracts the appropriate geometry from the quadtree for the current frame. The recursion process traverses the quadtree in a depth-first fashion, and is applied to each quadrant of the current level unless the quadrant met the terminating condition, in which case the vertices in that quadrant are used to compose OpenGL triangle fan primitive for rendering. The crude terminating condition we used is whether the ratio between the size of the quadrant block and the distance from the center of the block to the view point exceeds a predefined threshold. If the ratio is greater than the threshold, the quadrant will be further divided, otherwise it will be rendered. The intuitive reasoning of this condition is that when the ratio is lower than the

threshold, it means that the block in question is too far from the viewing point, or the dimension of the block is too small to affect the visual quality significantly, therefore further subdivision is redundant.

While this naive version of the terrain rendering constructed the quadtree and implemented the dynamic extraction of the mesh geometry, it has some detrimental characteristics. First of all, the visual jumps between the LODs are too blunt. Because discrete LODs are represented by the nodes on different levels in the quadtree, the edge vertices encoded in a quadtree block are either all rendered or all skipped. This created a obvious visual gap between the terrain region rendered with a finer LOD and the adjacent regions rendered with a coarser LOD. Secondly, the difference between detail levels cause the problem of cracks or T-joins. It is due to the inconsistency between a finer-detailed quadrant or its sub-quadrant and a coarser-detailed neighboring quadrant, as seen in Figure 4.1. These inconsistencies will form holes in the terrain surface, undermining the visualization. Therefore we continued our research and implementation to solve these shortcomings. In the next two sections, the solutions we found for the two problems mentioned above are described.

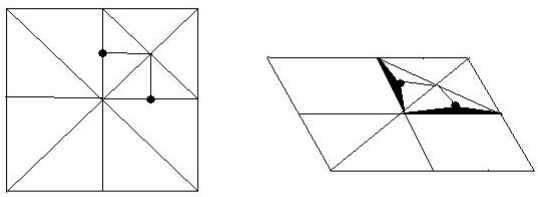


Figure 4.1: Quadtree inconsistency and crack artifacts

4.3.2 A More Sophisticated Approach

To solve the problem of steep visual transitions between levels, we adapted a more sophisticated approach to extract the geometry details from the quadtree data structure, and correspondingly render the geometry using OpenGL triangle-fan primitive

instead of quad-strips. We implemented the general framework for real-time adaptive quadtree LOD described in the paper of Peter Lindstrom et. al. [5]. The work in this paper was extended in a technical article by Thatcher Ulrich published on gaming development site Gamasutra [27]. We referred to his article for some implementation details and used some of his terminology in this paper. In this framework, the procedure for rendering the appropriate LOD is composed of two steps, updating and rendering. The creation of the quadtree is done once offline, and the updating and rendering are executed for each frame, as the algorithm is real-time.

Updating Vertex Flags for Rendering

The purpose of the updating step is to find which subset of the vertices in the quadtree will be rendered to form the mesh surface for the current frame. Before discussing further details of the updating process and criteria, we need to take a look at the specific structure of the quadtree blocks and the relation between the parent and child block.

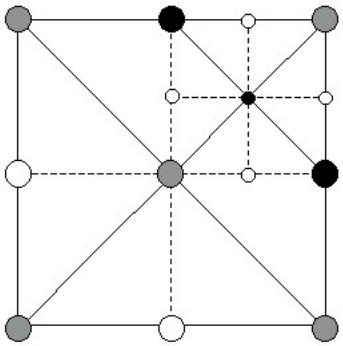


Figure 4.2: Quadtree detailed structure

As shown in Figure 4.2 above, each quadtree block has 9 vertices in a 3×3 formation. Except one vertex in the middle, the remaining 8 vertices are on the border of the block, with 4 on the corners and 4 on the edges. If we connect the central vertex and the four edge vertices, the block is divided into four quadrants. For the quadrants needed to be further subdivided, each of them is a child block of the current block, and the four corner vertices of a child block coincide with the

central vertex, one corner vertex, and two edge vertices of the parent block.

We adapted a top-down approach for updating the vertices. Starting from the entire terrain, which is represented by the root in the quadtree, the algorithm recursively subdivides the mesh and visits the corresponding quadtree nodes in a depth-first order. The algorithm uses labeling to record which vertices will be rendered. If one vertex needs to be rendered, we label it as "enabled". When the algorithm decides to further split a quadtree block, the block is "enabled" by enabling its central vertex and four corner vertices. For the side vertices not covered by enabled children blocks, a vertex error check is done to determine whether skipping this vertex in rendering will have significant visual impact. The aim of this check is to compare the height of the original vertex and the linear interpolated point at this location. The distance between the original vertex and the interpolation from the two adjacent corner vertices is defined as vertex error, as shown in Figure 4.3.

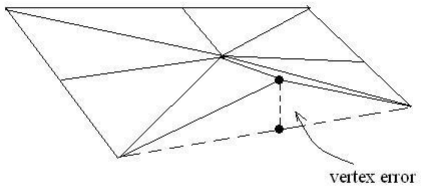


Figure 4.3: Quadtree vertex error

As proposed in the article by Ulrich, we formulated the check as the ratio between the vertex error and the approximated distance from the vertex to the viewing point. Let the camera location be (x_c, y_c, z_c), and the vertex location be (x_v, y_v, z_v). Define E_v to be the vertex error. We compute the error ratio R as:

$$R = \frac{E_v}{\sqrt{(x_v - x_c)^2 + (y_v - y_c)^2 + (z_v - z_c)^2}} \tag{4.1}$$

If error ratio R is less than a predetermined error threshold τ, then the visual

difference between the original vertex position and the interpolated approximation is not significant enough to justify the rendering of the vertex; otherwise, the vertex will be enabled.

The question remains how the algorithm decides whether a quadrant of the current block should be further divided, which is the point recursion happens. Similar to the vertex error testing, a "box test" termed by Ulrich is carried out. In a box test, the ratio compared to the error threshold depends on three variables: the size of the quadrant, its distance to the viewing point, and the maximum vertex error value included in the current block and all sub-level blocks encompassed in the quadrant. We can compute the ratio as following:

Let the coordinate for the center vertex of a block be (x_b, y_b, z_b), and s be the length of the block edge, then

$$R_{box} = \frac{E_{max}}{\sqrt{(x_v - x_c)^2 + (y_v - y_c)^2 + (z_v - z_c)^2} - s} \tag{4.2}$$

where the E_{max} is the maximum error contained in the current block and all its sub-blocks, which is identified during the quadtree construction. If a quadrant has a ratio R_{box} smaller than τ, it indicates that the maximum error caused by interpolation is still not enough to produce significant visual difference. Therefore, any further subdivision in this quadrant block will be unnecessary, and this quadrant will be rendered as a part of its parent block.

Rendering Quadtree Mesh

After the updating step, we have enabled all the vertices that need to be rendered. As the second step, the algorithm triangulates all the enabled vertices and renders the terrain mesh. It is actually straightforward because the vertices are organized into quadtree nodes. As each node in the quadtree references to the vertices forming its block region, the rendering can be achieved by a depth-first traversal in the quadtree.

As each node is a 3×3 vertex formation, it can be rendered by a predefined OpenGL primitive called triangle fan. However, there are some complications. First,

if a quadrant of the quadtree block will be further divided, it should not be rendered in the triangle fan covering the current block. Because the triangle fan must be continuous, a subdivided quadrant appears to interrupt the block to be covered by a single triangle fan. Secondly, if the side vertex between two quadrants without further subdivision is not enabled, the two neighboring corner vertices will form a single triangle with the vertex in the center. Thus, the formation of the triangles in a triangle fan depends on which quadrants are subdivided. Different cases of the triangle fan formations are illustrated in Figure 4.4.

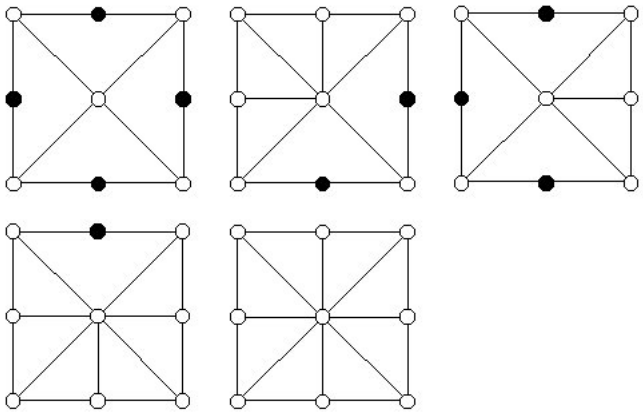


Figure 4.4: Triangle Fans with Different Vertices Enabled

Initially, we render the quadtree with exactly four triangle fans, each occupying one quadrant. Depending on whether the quadrant is subdivided, the triangle fan will make sure the quadrant area and the boarding areas are filled. However, with each node in the quadtree costing four triangle fans, the recursive rendering quickly became very expensive. After some consideration, we improved the rendering process for each quadtree block so that only one triangle fan is drawn for each block. The method we used is to define special triangles where the quadrant should not be rendered by the triangle fan. When the excluded quadrant is reached, the center vertex is supplied to the triangle fan definition process, so that a triangle with two identical vertices is added into the triangle fan. This special triangle keeps the continuity of the triangle fan but does not interrupt the geometry, because the whole triangle contracts to a line segment and coincides with one edge of the quadrant sub-block. The other side of the quadrant can be managed in the same way, and normal triangles then can be added into the connected triangle fan.

4.4 Dealing with Crack Artifacts

As mentioned in the first section, another severe problem with the rendering procedure is that it generates a large amount of artifacts in the rendered scene. The reason is that if a more detailed LOD is used for one quadrant but not for an adjacent one and the edge vertices for the more detailed LOD are enabled, interpolated approximation is used at the position of the edge vertex in the less detailed LOD. The inconsistency in geometry will cause holes and cracks in the rendered terrain surface.

In their 1987 paper, Brian Von Herzen and Alan H. Barr introduced the concept of restricted quadtree to overcome the problem of cracks [8]. This is further extended by Peter Linstrom et. al. in their paper about real-time adaptive LOD [5], where they proposed to represent the restricted quadtree requirements as a vertex dependency graph. With this vertex dependency, any vertex that is on the dependency path from the enabled vertex must be also enabled. We implemented a simplified version of the vertex dependency graph by storing references to all the vertices one vertex depends on in the vertex object we defined for each vertex in the mesh. In the original paper, a central vertex only depends on two of the corner vertices [5]. As seen in Figure 4.5, we defined the central vertex as the dependent of all four corner vertices of that block, in order to avoid checking different cases in which only two vertices are used.

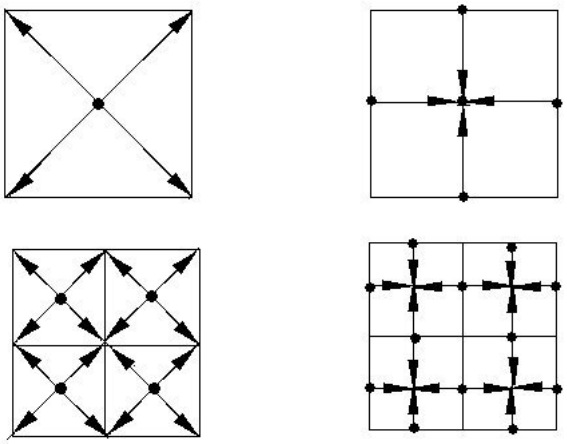
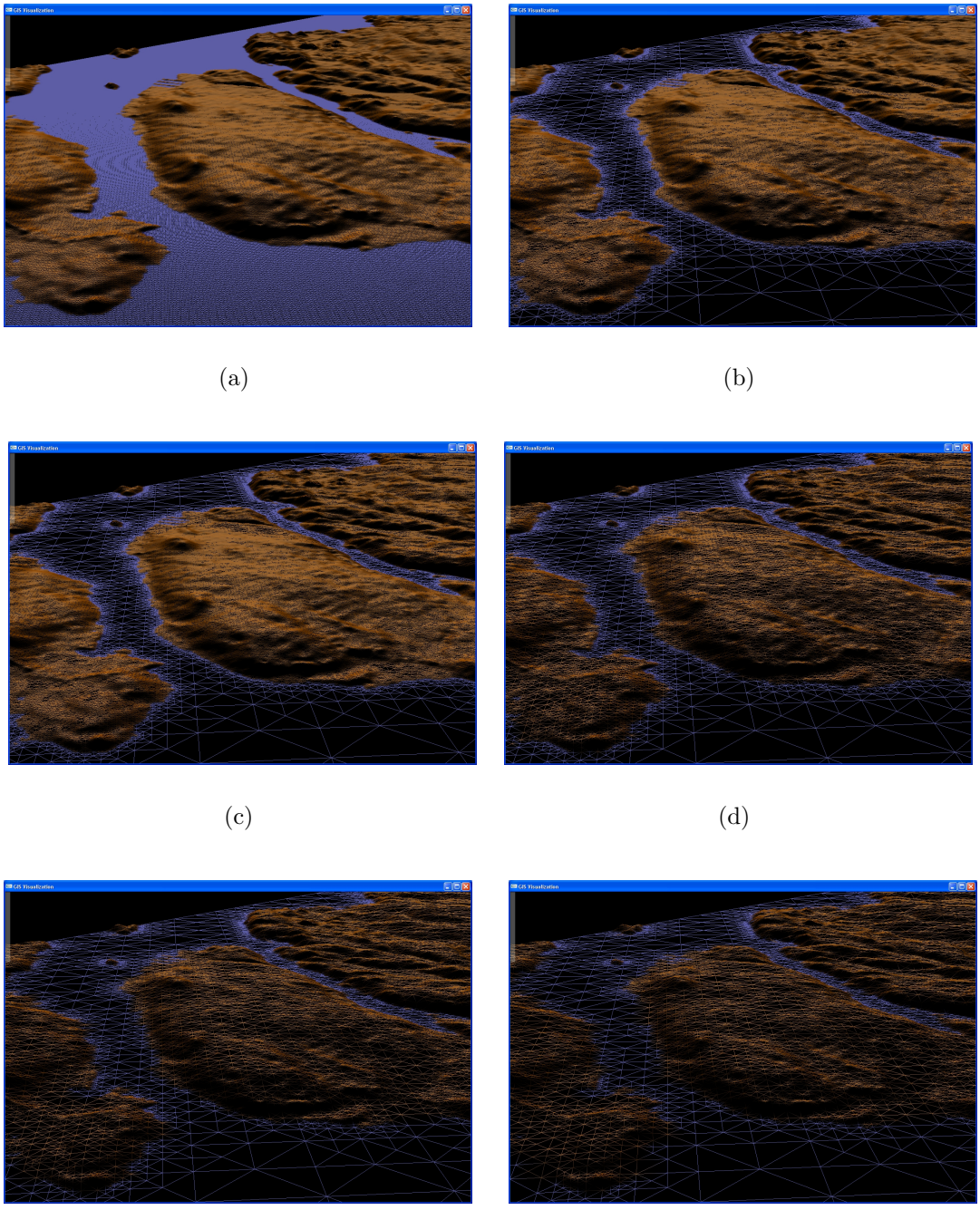


Figure 4.5: Vertex Dependency

Our approach to eliminating the cracks is to add an intermediate step after the updating and before the rendering. If a list of all possible vertices that may introduce the crack artifacts is kept, we can go through it in the intermediate step and make sure the vertex dependency is enforced. As determining whether a vertex introduces artifacts during the updating process is hard, we stored all the edge vertices we enabled. The first thing we tried was recording these vertices by putting their indices into a container. But with the large number of vertices we need to record, the insertion operations introduce too much overhead. So we decided to use a boolean variable array where each value associates with a vertex in the mesh. When an edge vertex is enabled, we turn on the flag in this array, indicating this vertex potentially causes crack artifacts. Then in the crack-eliminating step, we find each edge vertex and recursively turn the vertices which the current vertex depends on. Although large number of edge vertices do not cause artifacts, processing them does not cause too much overhead, because the vertices they depend on must have been enabled, which is the exact reason they don't introduce a crack. Wire frame images of our terrain LOD with different threshold values are listed below in Fig 4.6.



(a) (b)

(c) (d)

(e) (f)

Figure 4.6: Wire Frame LODs with Different Thresholds: From top left to bottom right, the images are original geometry, LODs with threshold 0.0001, 0.0005, 0.001, 0.0015, 0.002.

Chapter 5

Optimization

As outlined above, we implemented a simplified version of a real-time adaptive quadtree rendering framework. Although our program functioned properly and was able to render the crack-free terrain with different LODs depending on the vantage point, the implementation was still quite inefficient in terms of both the high-level rendering algorithm and the low-level programming techniques. In this chapter, we will discuss the work we did to optimize our basic implementation.

5.1 Optimization of Code Details

Due to the nature of the algorithm, most of the process described in the algorithm will be executed once for each frame. A slight difference in the execution time of a routine may often make a significant impact on the overall running speed of the program. Furthermore, with the relatively large size of our input data, any program modification to reduce the running time is justified, even if the improvement of the run time for a program iteration is minute. With this goal in mind, we investigated and found several minor time-saving modifications.

First, we reduce the number of function calls to change material colors. To simulate the interaction between light and materials, OpenGL defines a set of properties for each material in terms of how it reacts to lights. Normally, all the property settings are changed when a material is changed to another one, and four OpenGL function calls are needed to make the switch. Originally we have different material definitions for the land and water. Because the most obvious change is the color and finer lighting details are hard to detect viewed from a distance, we decided to reduce the number of function calls by only changing the color between brownish yellow for

the land and blue for the water body. With fewer function calls, the rendering is slightly faster. Similarly, we used a parallel light as the light source for the scene, because it is faster to compute shading effects with a parallel light source compared to a point one.

Along similar lines, we made further improvements in terms of general programming practice. As mentioned above, we use boolean flags to indicate potential crack-inducing vertices. For some flags, we clear them immediately after use, saving a separated iteration of the data set to reset them. For the flags that can not be reset that way, a separate iteration is carried out to reset them. We discovered that accessing a boolean array element is much faster than updating their values, so we check the flags first and only reset the ones necessary.

5.2 Quadtree Culling

After implementing the basic structure of my quadtree terrain rendering engine, the frame rate was still not satisfactory. After some research, I realized one possible place where I can improve the performance of it is to add in quadtree culling.

Initially, the quadtree update and rendering procedures are applied to the entire terrain. Starting from the center of the terrain, it recursively processes each of the four quadrants, if they meet the criteria in the box tests. But there are a lot of unnecessary computations, because unless the viewing angle is adjusted such that most of the terrain is visible, a lot of the updating and rendering computations won't contribute to the final projected image on the screen, because that part of the terrain is outside of the viewport, OpenGL is automatically culling those parts out. Therefore, if we can do a conservative estimation about how much of the terrain is not to be rendered, we can cut down the execution cost of the quadtree rendering process.

5.2.1 General 2D Clipping Principle

OpenGL determines which portion of the 3D space is visible by defining a viewing volume in front of the virtual camera. The surfaces of this viewing volume are the clipping planes, and only the objects appear inside the viewing volume are rendered onto the view port. So, in order to figure out which part of the terrain will be rendered, we need to find the region of the terrain surface included in the viewing volume.

As described previously, the quadtree represents a geometric hierarchical structure in 2D. For our application, the terrain surface is recursively subdivided in $x - z$ plane to form the LOD levels. To make the rough estimation about the regions to be clipped, it is sufficient to consider the horizontal positions in $x - z$ plane, regardless of the elevation in the y dimension. Therefore, the problem we need to solve for estimating terrain clipping is to determine whether a projection of a vertex will fall in the region inside the viewing volume, and is transformed to a problem in the 2D domain.

5.2.2 Clipping Line Equations

How can we figure out the region where the viewing volume intersects with the $x - z$ plane? As we are calculating an estimation of the intersected region rather than the exact dimensions, a bounding rectangular was adapted to approximate the region.

To find the intersecting area, we need to make an approximation based on the shape of the viewing volume. There are two types of view volumes in OpenGL: a rectangular box defined by orthographic projection and a truncated pyramid defined by perspective projection, as shown in Figure 5.1. In our terrain engine, the perspective projection is used to produce more realistic images. To simplify implementation, we defined the bounding rectangle as following: extending from the camera, the length of the bounding box is the distance from the $x - z$ plane projection of the camera position to the projected line of the far edge of the top clipping plane, as seen in Figure 5.2. We used this approximation with the assumption that the orientation of

the camera will not be higher than the horizontal direction, so that the far edge of the top clipping plane on the viewing pyramid will always have the largest distance from the view point. For the width of the 2D clipping region, we used the width of the far clipping plane of the truncated pyramid, because it is the widest plane in the viewing volume. Therefore, using a set of variables describing the viewing volume, in our case a truncated pyramid, the lines in $x - z$ plane that form the bounding rectangle for our clipping area can be expressed.

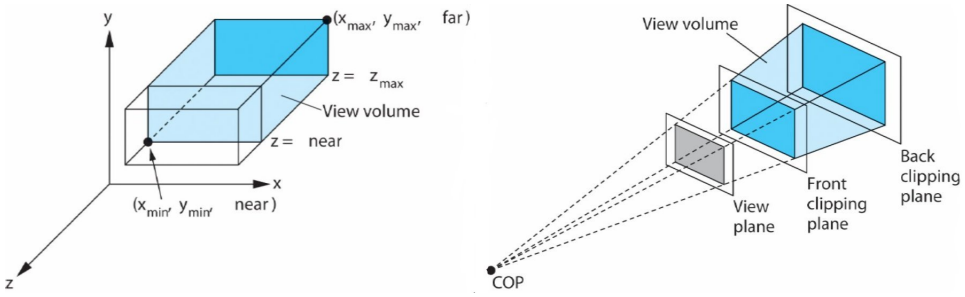


Figure 5.1: OpenGL Viewing Volumes: orthographic viewing (left) and perspective viewing (right) [28]

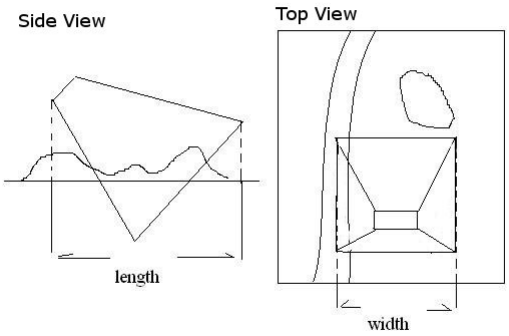


Figure 5.2: Dimensions of Clipping Area with Viewing Volume Projection

In OpenGL, the truncated pyramid viewing volume is defined by several parameters: the field of view fov, which is the angle between the top and bottom clipping plane in the viewing volume. Let z_{near} denote the distance from the camera to the near clipping plane, and z_{far} denote the distance to the far clipping plane. Also, define *aspect* as the aspect ratio of the far vertical clipping plane. As mentioned previously, we keep track of the location and the orientation of the virtual camera by storing the angles that the virtual camera needs to rotate from the standard position.

More specifically, the location of the camera in the world coordinate is denoted as (x_C, y_C, z_C) , and define θ_x and θ_y to be the angles by which the camera needs to rotate from the initial orientation. Note that the rotation along the x-axis is done before the rotation along y-axis, so that both rotations are done using standard axes. With the variables mentioned above, the clipping rectangle is bounded by four lines $L0, L1, L2$, and $L3$, and their equations can be expressed as following:

$$\begin{aligned}
\tan(\theta_y)x - y + z_C - x_C \tan(\theta_y) &= 0 \quad (L0) \\
-\cot(\theta_y)x - y + (x_C - \cos(\theta_y)\tfrac{width}{2})\cot(\theta_y) + z_C - \sin(\theta_y)\tfrac{width}{2} &= 0 \quad (L1) \\
\tan(\theta_y)x - y + z_C - x_C \tan(\theta_y) - \tfrac{z_{far}}{\cos(\frac{\text{fov}}{2})}\tfrac{\cos(\theta_x)}{\cos(\theta_y)} &= 0 \quad (L2) \\
-\cot(\theta_y)x - y + (x_C + \cos(\theta_y)\tfrac{width}{2})\cot(\theta_y) + z_C + \sin(\theta_y)\tfrac{width}{2} &= 0 \quad (L3)
\end{aligned}$$

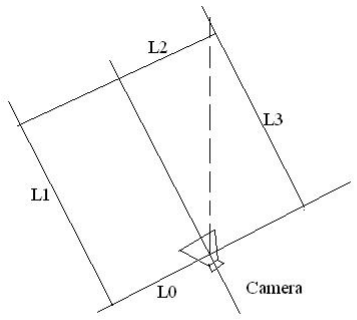


Figure 5.3: Clipping Rectangle Bounding Lines

5.2.3 Applying Clipping to Quadtree

Now that the bounding rectangle for the terrain culling is known, we need to utilize this during the quadtree rendering process. The aim here is to reduce the unnecessary calculation when the program traverses through the quadtree. As we mentioned above, the process to render the quadtree has two parts, namely updating and rendering. If the quadtree traversal is pruned during the update process, the portion of quadtree laying outside the viewport will not be enabled during the updating process,

which means they will not be rendered in the second step of the process. Thus, it is natural that we integrated the clipping into the updating step.

In the updating step, a depth-first traversal is carried out. For a given block in the quadtree, the updating routine will be recursively applied to one or more of its children which pass the box test. We added the clipping check here so that the clipping result of the child block determines whether the recursion will continue. If the four vertices forming the corners of the block are all clipped, the child block is clipped, and it is not necessary to further update its children. When at least one of its corners is included in the viewing volume, part of the region in the block needs to be rendered, and a further split is needed.

After adding the clipping checks before each deeper recursion will be started, regions of the terrain not in the camera field of view were quickly pruned from the updating recursion. This potentially saved updating operations on hundreds of vertices, and along with their rendering time. With the amount of rendering proportional to the number of vertices underlying the partial terrain appearing in the camera view, quadtree culling further improved the computational efficiency of the terrain mesh, making the rendering process output-sensitive.

Chapter 6

Terrain Collapsing

Working with a large terrain model, the user often needs to visually compare two patches on the terrain surface. With traditional 3D terrain engines, the virtual camera must be navigated from one terrain patch to the other one for comparison; alternatively, the user can zoom out to enclose the two patches of terrain into one screen, with the expense of specific details. We designed terrain collapsing as a dynamic terrain manipulation tool to solve this problem. Given two terrain patches specified by the user, the goal is to bring the two areas of interests together so that they are displayed in the same screen. Because the terrain region between them occupies much less area as it originally does, we named this functionality "terrain collapsing", and this middle region "collapsing zone".

The techniques we used to achieve the collapsing effects evolved through different development phases of our application, and changed when we used the adaptive quadtree as our underlying mesh structure. This is because the method to collapse the terrain depends closely on the underlying mesh structure. When we used the simple quad-strips to form the mesh, the possible collapsing effects are restricted by the geometry structure, and the method is specifically designed for this scenario. Later we adapted the real-time LOD quadtree mesh, the method of terrain collapsing was no longer applicable, because the regularity of the mesh structure no longer exists. However, the new mesh structure also gave us greater flexibility. For example, the collapse can now be done in arbitrary user-specified orientation instead of the fixed directions along the axes. We will briefly discuss the collapsing with the quad-strips in the following section and the collapsing with quadtree LODs in the section after that.

6.1 Collapsing Quad-Strip Meshes

As mentioned in the chapter about meshing, we initially modeled the terrain by a large set of parallel quad-strips, where each strip is formed by quadrilaterals concatenated one after another. Formed from the regular elevation data grid, the quad-strips are defined aligned to the x axis. Taking advantage of the inherited division of the model, we restricted the collapsing in only two orthogonal directions, namely along x axis and along z axis.

As described in Chapter 3, the user can specify a point on terrain surface by mouse picking. Once a point is selected, depending on the current collapsing state, one boundary of the collapsing zone in the x or z direction can be determined. The other boundary of the collapsing zone is determined in the same way. Along the collapsing axis, the whole terrain is divided into three parts: the collapsing zone and the two non-collapsing zones on its two sides. The collapsing was achieved by batch rendering and 3D transformations. First, one non-collapsing zone is rendered, and a scaling transformation is done before the collapsing zone is rendered. With the transformed Modelview matrix in OpenGL, the collapsing zone is shrunk, followed by the other non-collapsing zone. Besides the simplicity in the implementation, scaling down the terrain preserves the geometry and texture of the zone, therefore providing the user with a general sense of how much of the terrain is cramped into the collapsing zone. Because the quad-strips spanned across the entire terrain along the x axis, collapsing along the z axis is slightly more complicated, because the zones are composed of partial geometric objects instead of complete ones. So one previous strip needs to be rendered as three adjacent segments, falling into the three zones separately.

One thing worth noticing is that, regardless the orientation of the collapsing, the effects can be achieved because of the regularity of the mesh vertices. As the vertices distribute uniformly according to the data points in the height field, the location a user specified can always be mapped to a single cell in the mesh, enabling the zone partitions.

6.2 Collapsing Quadtree Meshes

Collapsing in a quadtree mesh differs significantly from that in the regular-grid mesh. In an adaptive LOD quadtree, the high regularity no longer exists. As the vertices are enabled according to the local geometry, the vertex density of the mesh will vary from very high at regions with complex geometry, to very low at more flat regions. Thus, it is difficult to group vertices into discrete geometric objects. More importantly, the quadtree LODs are rendered by a recursive routine, making the procedural approach to collapsing inapplicable here.

To adapt to this new situation, we implemented terrain collapsing with a new method. Instead of doing modelling transformations to scale terrain zones, we manually calculate a temporary new coordinate for each vertex and render the terrain using these temporary coordinates. As the elevations of the data points are irrelevant, we denote the horizontal plane as x-y plane to simplify the presentation. We derived the "transformed" coordinates as following:

Given 2 target points A with coordinates $(x_1, y_1), B(x_2, y_2)$ by mouse clicking; the middle point can be calculated: $M(x_m = \frac{x_1+x_2}{2}, y_m = \frac{y_1+y_2}{2})$

For the point $P(x', y')$ we want to transform, try to find the closest point $Q(x, y)$ on the line l_n that passes C and is perpendicular to $\overrightarrow{AB}$, as shown in Figure 6.1. Let k_{AB} denote the slope of $\overrightarrow{AB}$.

$$\because PQ \perp QC, k_{AB} = \frac{y_1 - y_2}{x_1 - x_2};$$

$$\therefore \frac{y - y'}{x - x'} = k_{AB} \tag{6.1}$$

$$\because QC \perp AB$$

$$\therefore \frac{y - y_m}{x - x_m} = -\frac{1}{k_{AB}} \tag{6.2}$$

solve (6.1)&(6.2) :

$$k_{AB}(x - x') + y' = -\frac{1}{k_{AB}}(x - x_m) + y_m$$

$$\therefore \begin{cases} x = \frac{k_{AB}x' + \frac{x_m}{k_{AB}} + y_m - y'}{k_{AB} + \frac{1}{k_{AB}}} \\ y = k_{AB}(x - x') + y' \end{cases}$$

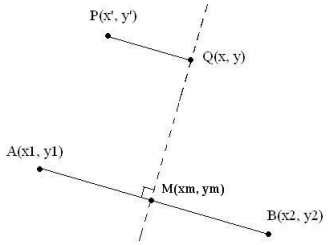


Figure 6.1: Calculate the Collapsing Center

Therefore, for any vertex P, we can find its projection Q on the line perpendicular to $\overrightarrow{AB}$ through point C. Then, depending on whether point P is outside the collapsing zone, its new coordinate after the collapse can be derived as following: If P is inside the collapsing zone,

$$\begin{cases} x_{collapse} = x + r(x' - x) \\ y_{collapse} = y + r(y' - y) \end{cases};$$

where r is the collapsing ratio. Otherwise, the point is outside the collapsing zone, and only needs to be translated to the approperiate location. From the trigonometric relations:

$$\Delta x = \cos(arctg(k_{AB}))(1 - r)d$$
$$\Delta y = \sin(arctg(k_{AB}))(1 - r)d$$

where d is the distance between P and Q. Therefore, the new coordinate for a translated vertex is:

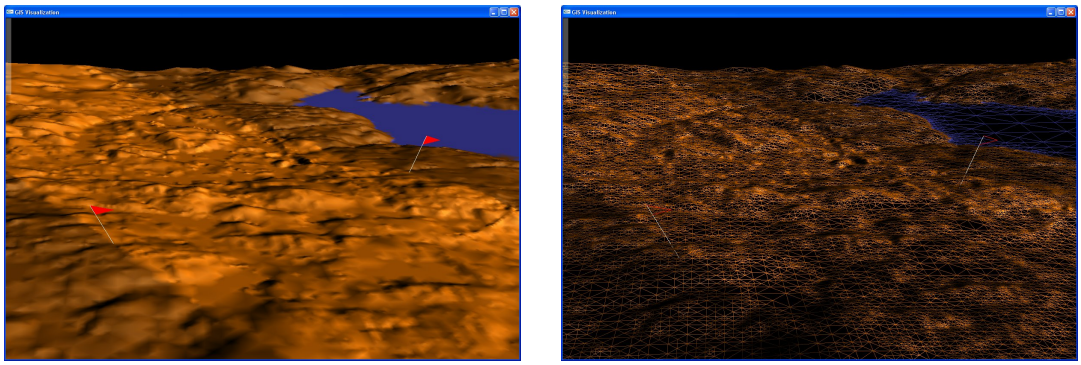
$$\begin{cases} x_{collapse} = x' \pm \Delta x \\ y_{collapse} = y' \pm \Delta y \end{cases} .$$

The specific values for the coordinate expressions above depend on the relative position of point P regarding point Q.

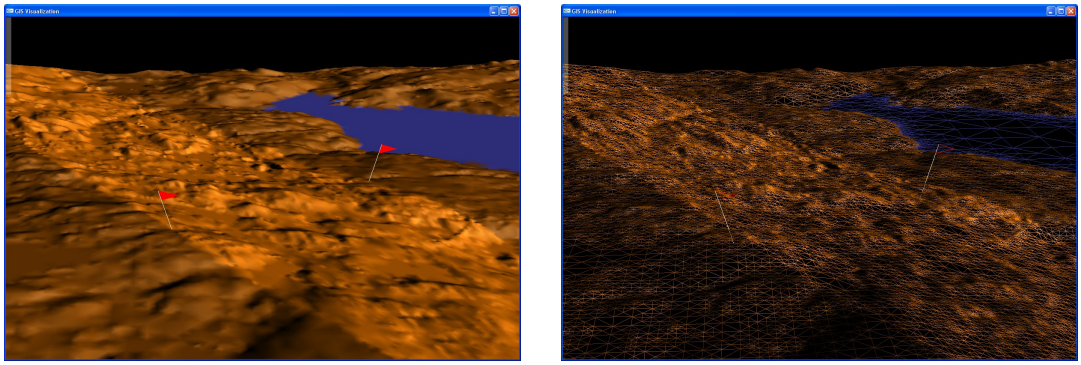
With these mathematical operations, each vertex in the 2D plane can be dynamically transformed around the collapsing zone specified by user input. Because the transformations involved in the terrain collapse are all horizontal, the elevation values for all vertices stay the same, therefore we only need to substitute the x and z coordinates of a vertex into the expressions above, in order to get the new location of that vertex in the $x - z$ plane.

As mentioned previously, we adapted this method of manual transformation only when the underlying geometry was changed to adaptive quadtree and the previous method became inapplicable. However, the new geometric structure provided greater flexibility in terms of the collapse orientations. While with the quad-strips collapsing is only allowed along the two orthogonal axes, directional limitations no longer exist with the new collapsing method. As the calculations are formulated for a general collapsing axis, the user can collapse the terrain along an arbitrary direction by setting the proper input points.

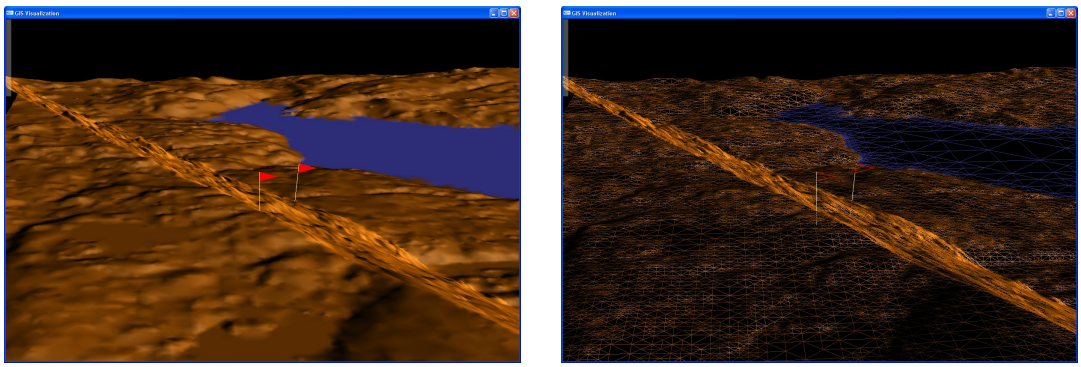
In terms of performance, although the new collapsing method requires additional calculation done for each vertex, this will not have severe impact on the overall performance because the total number of vertices needed for each frame has been reduced by the LOD implementation. As the scene further away from camera uses coarser LOD level, the total number of vertices needed for a frame are reduced. So, even though rendering individual vertices takes longer, the performance of the application is not compromised. The process of the terrain collapsing is shown in Figure 6.2 below.



(a) (b)



(c) (d)



(e) (f)

Figure 6.2: Terrain Collapsing Shown in Normal and Wire Frame Rendering

Chapter 7

Results

To quantitatively understand the performance of our implementation, we did some simple experiments with the terrain engine we implemented, and we describe the experimental setup and provide brief analysis in this chapter.

For testing purposes, our terrain model is generated with a 1024×1024 DEM data file for Halifax region. We fixed the virtual camera at a location from where the Halifax Penisula is viewed from the east. As we can see in Figure 7.1, the scene covered different terrain surface features, including the geometrically complex hilly area and some flat water body. The system we used for the experiments has Pentium Core 2 Duo 2.13GHz CPU, 2GB RAM and GeForce 8800GTX video card with 640MB on-board memory.

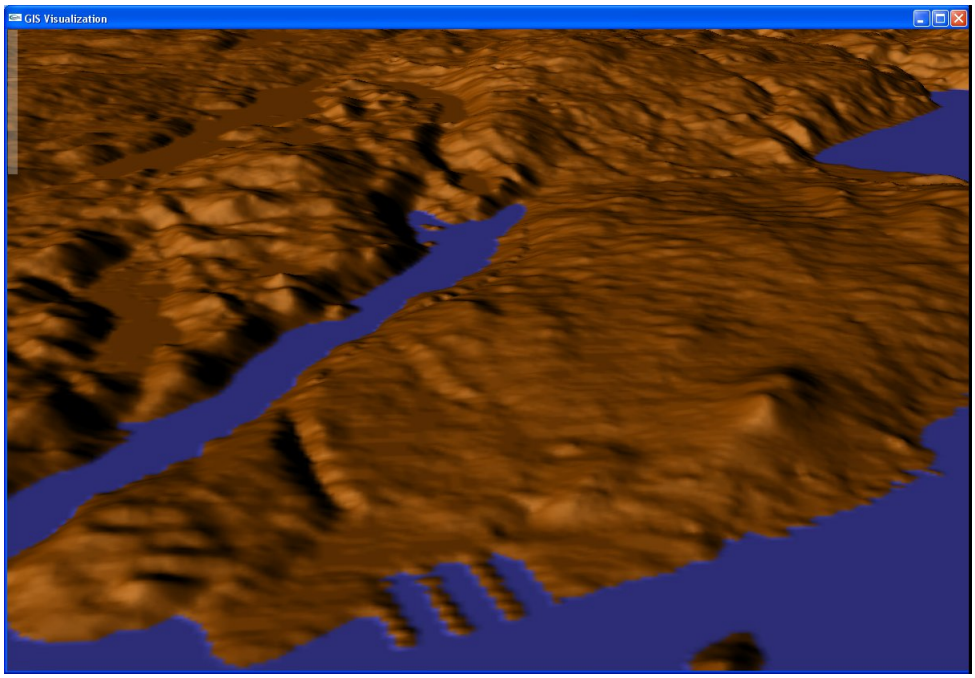


Figure 7.1: Scene Selected for Experiments: Halifax Peninsula

First, we tried to find out how the variation of the threshold value τ affects the frame rates and the geometric complexity. We built in an input control so we could manually change the value of τ. For the same fixed scene we mentioned above, changing the threshold value from 0 to 0.003 results in the variations of FPS values and the total number of triangles plotted in Figure 7.2.

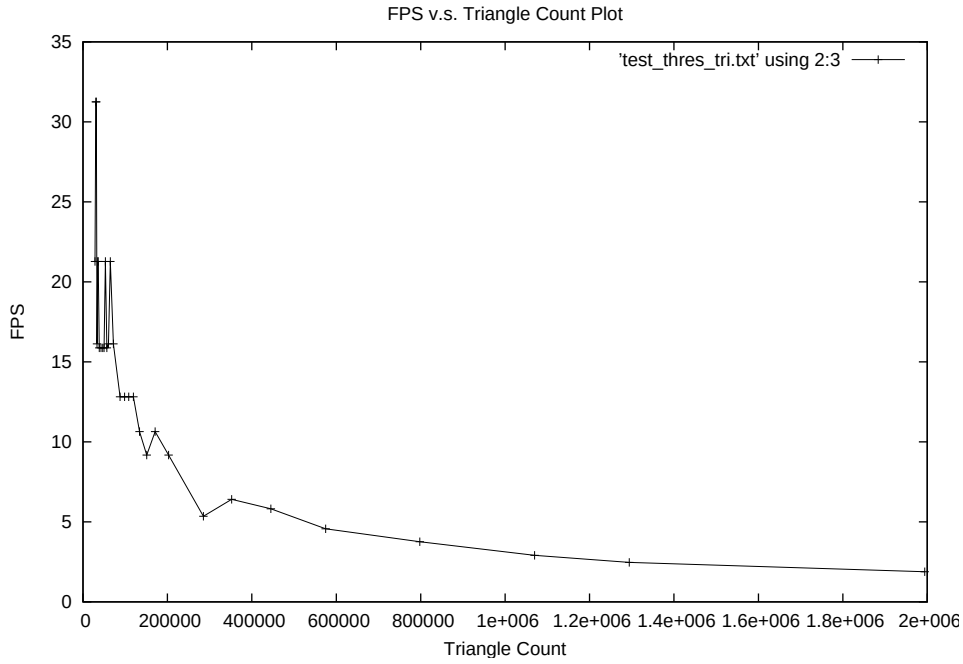
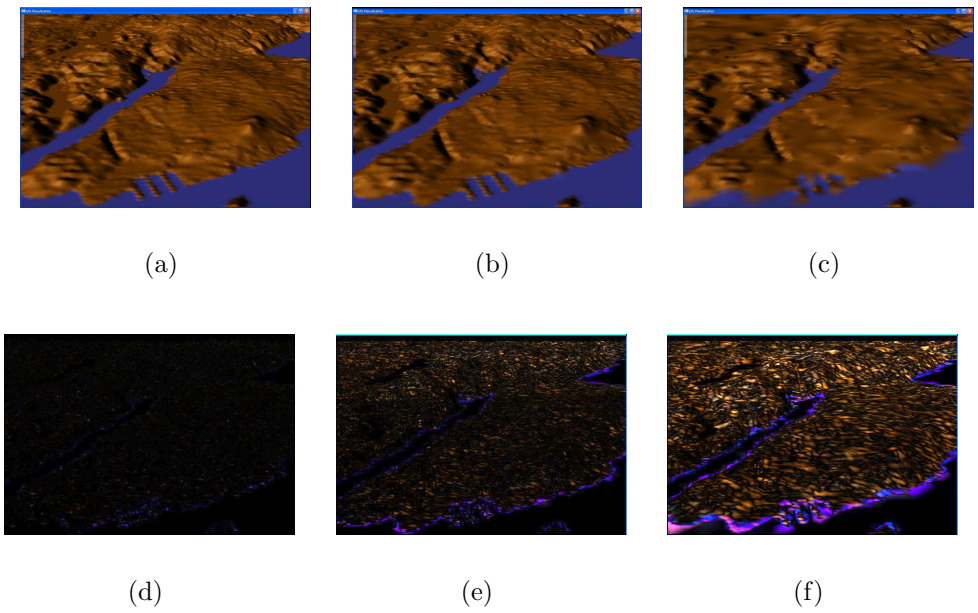


Figure 7.2: Experiment Result Plot 1: FPS v.s. Triangle Count

As confirmed in this graph, with fixed system resources, a frame with more complex geometry generally takes more time to render, causing a lower frame rate. However, the frame rates did not appear to increase linearly with the decreasing geometric complexity. After some closer observations of this behavior, we found out that the reason of this is how LOD works. Initially, when the threshold value first started to increase, the terrain regions further away from the camera immediately failed the enabling test, therefore were pruned from the current adaptive quadtree; but the terrain close to the camera still kept most of its geometry. Only when τ value is large enough to simplify the geometric complexity of the near-by regions, the frame rate began to increase sharply. Also, we suspect the localized zig-zag pattern in the graph is due to interruption from other processes running on the system.

As described in the section about LOD techniques, the goal of applying LODs

is to balance the visual fidelity and the speed of the graphics rendering. To test our implementation of the LOD framework, we wanted to see how the frame rate interacts with image fidelity. For a simple fidelity measure, we adapted root mean squared error (RMS error) [24], which is the sum of squared pixel value difference over all pixels. For the same scene mentioned above, the real-time images with different threshold settings were captured using GNU Image Manipulation Package (GIMP). As shown in Figure 7.3, these static images are compared to the image of the scene with full details, and the pixel values of the differential image are used to compute RMS error. The resulting plot is shown in Figure 7.4, where x axis is for RMS error, and y axis is for FPS values.



(a) (b) (c)

(d) (e) (f)

Figure 7.3: Testing and Error Images for Experiments: Top row is test image of a fixed scene, bottom row is their error image against the full geometry image; from left to right, the threshold values are 0.0001, 0.001, 0.003.

This plot depicts the trade-off between visual fidelity and the frame rate graphically. With greater threshold values, the simplified geometry for terrain mesh resulted in increased image error, whereas it also leads to increased frame rate. The wide jump in the error measure is mapped to the abrupt changes in the previous graph, and can be explained similarly. Under the assumption that RMS is a good measurement for visual fidelity, our LOD implementation can work better when the error is less than

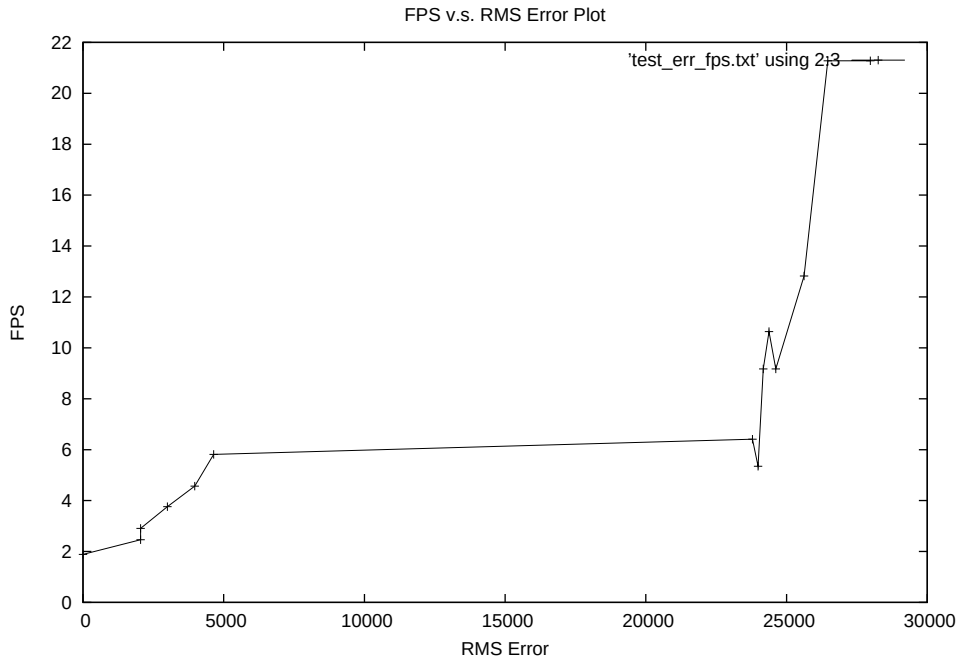


Figure 7.4: Experiment Result Plot 2: FPS v.s. RMS Error

5000 or around 25000, where it can improve the frame rate with moderate visual quality loss.

Chapter 8

Conclusions

We have designed and implemented a 3D GIS system with collapsible terrain, a GIS visualization application providing 3D terrain rendering capabilities and a new user control method we called "terrain collapsing". We implemented this user control feature, so that a GIS user can bring two far apart terrain regions into the same screen by temporarily changing the shape of the terrain. To improve the system performance, we also explored the efficient data structures and rendering techniques for terrain visualization.

Based on the topographic information in DEM format, our application models terrains by using a polygon surface mesh. As the full resolution geometry details are redundant and not suitable for interactive system, we adopted real-time LOD with a quadtree data structure, which has significantly improved the rendering efficiency of our system. With the OpenGL API, we have achieved realistic shading effects and advanced user interface controls. The functionality of terrain collapsing is implemented by calculating individual vertex coordinates according to their geometric relation with the collapsed region.

Our research and exploration in developing our system is an attempt to utilize the large potential of 3D GIS visualization. With the dramatic improvement of graphics capabilities in modern workstations, the research area of 3D GIS is getting more attention. In a more realistic 3D virtual environment, novel user control technologies will be demanded to facilitate more advanced visualization technology.

8.1 Future Work

There are several aspects of our system that may be enhanced. First, the current rendering routines are based on defining individual vertices, which is not as efficient as some more advanced techniques available in OpenGL. It is worth investigating how the different mesh rendering techniques affect the LOD techniques and the terrain collapsing implementation.

Also, the current implementation of terrain collapsing requires the coordinate calculation for each vertex to be rendered. Better methods to dynamically change the terrain geometry must exist. For example, the vertex shader may be suitable for this task, because vertex shaders use the computational power of the graphics card to control the position of individual vertices. Caching of the vertices may also be useful.

Furthermore, our system currently can visualize elevation data and satellite imagery. Adapting to new data formats, for example, vector GIS data, could greatly improve the power of the system.

Bibliography

[1] C. T. Silva, J. S. B. Mitchell, and A. E. Kaufman. Automatic generation of triangular irregular networks using greedy cuts. In *VIS '95: Proceedings of the 6th Conference on Visualization '95*, page 201, Washington, DC, USA, 1995. IEEE Computer Society.

[2] Robert J. Fowler and James J. Little. Automatic extraction of irregular network digital terrain models. In *SIGGRAPH '79: Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, pages 199–207, New York, NY, USA, 1979. ACM Press.

[3] Florian Schröder and Patrick Roßbach. Managing the complexity of digital terrain models. *Computers & Graphics*, 18(6):775–783, 1994.

[4] Michael Garland and Paul S. Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Sept. 1995.

[5] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 109–118, New York, NY, USA, 1996. ACM Press.

[6] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[7] R. Sivan and H. Samet. Algorithms for constructing quadtree surface maps. In *In Proc. 5th Int. Symposium on Spatial Data Handling*, pages 361–370, 1992.

[8] Brian Von Herzen and Alan H. Barr. Accurate triangulations of deformed, intersecting surfaces. In *SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pages 103–110, New York, NY, USA, 1987. ACM Press.

[9] Renato B. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 19–26, 1998.

[10] Carol Traynor and Marian G. Williams. Why are geographic information systems hard to use? In *CHI '95: Conference Companion on Human Factors in Computing Systems*, pages 288–289, New York, NY, USA, 1995. ACM Press.

[11] Ingmar Rauschert, Pyush Agrawal, Rajeev Sharma, Sven Fuhrmann, Isaac Brewer, and Alan MacEachren. Designing a human-centered, multimodal GIS interface to support emergency management. In *GIS '02: Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems*, pages 119–124, New York, NY, USA, 2002. ACM Press.

[12] Andreas D. Blaser and Max. J. Egenhofer. A visual tool for querying geographic databases. In *AVI '00: Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 211–216, New York, NY, USA, 2000. ACM Press.

[13] David Koller, Peter Lindstrom, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory Turner. Virtual GIS: a real-time 3D geographic information system. In *VIS '95: Proceedings of the 6th Conference on Visualization '95*, page 94, Washington, DC, USA, 1995. IEEE Computer Society.

[14] Stephen Brooks and Jacqueline L. Whalley. A 2D/3D hybrid geographical information system. In *GRAPHITE '05: Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, pages 323–330, New York, NY, USA, 2005. ACM Press.

[15] Y. K. Leung and M. D. Apperley. A review and taxonomy of distortion-oriented presentation techniques. *ACM Trans. Comput.-Hum. Interact.*, 1(2):126–160, 1994.

[16] I. TZAVARAS M. D. APPERLEY and R. SPENCE. A bifocal display technique for data presentation. In *Eurographics 82*, pages 27–43, 1982.

[17] Jock D. Mackinlay, George G. Robertson, and Stuart K. Card. The perspective wall: detail and context smoothly integrated. In *CHI '91: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 173–176, New York, NY, USA, 1991. ACM Press.

[18] Chong K. Liew, Uinam J. Choi, and Chung J. Liew. A data distortion by probability distribution. *ACM Trans. Database Syst.*, 10(3):395–411, 1985.

[19] G. W. Furnas. Generalized fisheye views. In *CHI '86: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 16–23, New York, NY, USA, 1986. ACM Press.

[20] M. Sheelagh T. Carpendale, David J. Cowperthwaite, and F. David Fracchia. Extending distortion viewing from 2D to 3D. *IEEE Computer Graphics and Applications: Special Issue on Information Visualization*, 17(4):42–51, / 1997.

[21] Farid Mamaghani. What makes virtual systems a reality. *SIGGRAPH Comput. Graph.*, 28(2):105–109, 1994.

[22] Thomas A. Funkhouser and Carlo H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *SIGGRAPH '93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pages 247–254, New York, NY, USA, 1993. ACM Press.

[23] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[24] David Luebke, Benjamin Watson, Jonathan D. Cohen, Martin Reddy, and Amitabh Varshney. *Level of detail for 3D graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[25] James Kaba and Joseph Peters. A pyramid-based approach to interactive terrain visualization. In *PRS '93: Proceedings of the 1993 Symposium on Parallel Rendering*, pages 67–70, New York, NY, USA, 1993. ACM Press.

[26] Renato Pajarola. Overview of quadtree-based terrain triangulation and visualization. Technical Report UCI-ICS-02-01, Jan. 2002.

[27] Thatcher Ulrich. Continuous LOD terrain meshing using adaptive quadtrees. Technical Report 20000228, Feb. 2000.

[28] Edward Angel. *Interactive computer graphics: a top-down approach with OPENGL primer package-2nd Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2001.