## Final Report

Landscape map/quadtrees; Game with 2D or 3D graphics, dynamics, networking


Cardiff university school of computer science

## Author: Jin Chen

Supervisor: Dr Frank C Langbein

One semester individual project

## Acknowledgement

In the end, I want to thank everyone who helped me during this time. First of all, I want to thank my supervisor Dr. Frank Langbein. His rich knowledge helped me solve many doubts and always pointed out many ways for me when I faced difficulties. Although I always couldn't keep up with the progress, he patiently comforted me, calmed my panic heart and cheered me up. It is my luck to meet Dr. Frank, and I will continue to move forward with gratitude.

I want to thank my family. They have given me unlimited tolerance and support during this time. I will continue to work hard to repay their expectations of me.

I want to thank my love Shaolyn. Her encouragement and support helped me through all the difficulties. My gratitude to her can no longer be expressed in words. I will use action to bring us a better life in return for everything she has done to me.

Thanks to all the researchers and developers, your efforts and results will forever be deposited in the hall of human knowledge. The task I am doing now is just standing on your shoulders and doing what you have done. I will be grateful for this.

## Table of Contents

1. Introduction ..... 5
1.1 Levels of detail ..... 5
1.2 The goal of this article ..... 6
2. Background ..... 8
2.1 Regular Square Grid (RSG) and Triangular Irregular Network (TIN) .....  8
2.2 Height Map ..... 9
2.3 Rendering ..... 10
2.4 Game Engine and API ..... 13
2.5 The Algorithms for Levels of Detail ..... 14
3. Visualizing landscapes with a quadtree ..... 18
3.1. Overview over the whole systems ..... 18
3.2 Rendering ..... 18
3.2.1 Vertex Data Transfer ..... 18
3.2.2 Camera ..... 20
3.2.3 Illumination ..... 22
3.3 Quadtree data structure ..... 24
3.4 Refinements ..... 27
3.4.1 Generation and storage of terrain data ..... 27
3.4.2 Elimination of objects outside the view frustum ..... 28
3.4.3 Segmentation of Quadtree Nodes ..... 29
3.4.4 Crack elimination ..... 31
3.4.5 Grid visualization ..... 34
4. Results and Analysis ..... 37
4.1 Crack ..... 37
4.2 Display of rendering results ..... 38
4.3 Simplification efficiency of terrain grid ..... 39
4.4 Compare the terrain with full details ..... 40
4.5 Compare different scenes ..... 43
5. Conclusions and future work ..... 44
6. Reflection ..... 46
References ..... 48

## 1. Introduction

Open world games, such as Genshin, Zelda, World of Warcraft, etc. The popularity of these games in the world not only caters to the aspirations of game lovers for the game world, but also opens the possible of the realization of the imaginary world which similar to the descriptions in Sword Art Online and other animations. On the other hand, the visualization of real landscape on a computer is an important skill whether in the Geographic Information Systems (GIS) or the military field. Therefore, countless scholars have worked hard in this field and made breakthroughs in all directions. This article focusses on the initial stage of this field, which is to visually render a continuous and broad landscape.

### 1.1 Levels of detail

Different from the real world, the visualization of landscape in the computer is achieved by using a large number of colored or textured polygon grids which can approximate the contour of curve of the terrain. In the displayed window, each figure is composed of many polygonal patches. When rendering one polygon, computer needs to obtain the position of each vertex in the polygon. When people want to visualize a complex and huge terrain, the computer will process a large number of polygons and vertices data. However, the performance of computer has its limits. This may require tens of millions of vertices information, which consumes a lot of computing resources and leads to the waste of it. Therefore, many developers have carried out research on the optimization of rendering, and it is also the content of this article.

When rendering objects, more triangle fragments can make the performance of rendering closer to the real objects. This also means that more computer resources will be consumed. On the other hand, when
detailed terrain information is not needed, it can also be achieved by reducing the number of triangles rendered. In this way, more computing resources can be released. As well, it is found that in real life, the closer the object is to the observer, the more details it will show; and when the distance is far away, the observer can only get the outline of the object. Applying this discovery to computer rendering, we can render objects with different details based on the distance between the object and the observer.

As a result, the rendering method - Levels of Detail (LOD) was created by developers. In this method, the triangular meshes that make up the terrain will become larger and lesser as the increased distance between the terrain and the observer. In order to implement this method, we need to render the landscape with different levels of detail at different distances and glue these levels of landscape to form a multi-resolution model (Mark 1997).

### 1.2 The goal of this article

This article will introduce some of the difficulty faced by LOD which been solved by this article:

1. How to construct terrain of different resolutions and realize the mutual conversion of these based on the given data of terrain. Since the observer is movable, we need to adjust the terrain resolution in real time according to the position of observer. Therefore, a real-time dynamic visualization of the levels of detail terrain is essential.
2. How to greatly reduce the number of polygons to be rendered while maintaining the quality of terrain visualization. The reduction in the number of rendered polygons will lead to the loss of image quality of the surface mesh. It is necessary to pay attention to maintaining the high quality of the image in the observer's perspective while optimizing the rendering grid.
3. How to smooth the joints between terrains of different resolutions.

Since higher-resolution terrain blocks have more vertices than lower one, these extra vertices may cause cracks at the junctions due to different elevation values. The appearance of cracks is not allowed in the visualization of landscape.

In this article, through the study and summary of previous research results, a quadtree algorithm that can solve the above problems is realized. Based on the rendering pipeline provided by OpenGL, this paper processes the terrain data, and realizes the visualization of the attributes such as the polygon mesh, texture, and basic lighting of the landscape. By storing the data of terrain in the quadtree algorithm, the conversion and determination of the terrain resolution are realized. At the same time, based on the location of the camera, the resolution levels of terrain at different distances are calculated in real time. In order to further improve the rendering efficiency, through the calculation of the view of camera, the rendering of all grids outside the angle of view is eliminated. In the end, it minimizes the impact on the image and releases a large amount of computing resources.

## 2. Background

### 2.1 Regular Square Grid (RSG) and Triangular Irregular Network (TIN)

In the computer, due to the different encapsulation methods, the map data has a variety of storage methods and structures. Usually, the terrain model data contains the information of all the points in the model, including the position and elevation value of the point (Emanuele 2006). By observing the intervals between these data points, these terrain model data can be divided into two main categories. The first type is called Regular Square Grid (RSG). As the name implies, these data points are regularly spaced and finally merged into a square grid. The terrain data stored in this way can be visualized only by storing the elevation value due to its regular attributes. This method not only greatly reduces the size of its data, but also improves its data utilization efficiency. However, RSG has shortcomings that cannot be ignored. It is also because of the regular attribution of data points that the terrain undulations between two data points cannot be taken when sampling the actual terrain, which leads to the inability to accurately describe the actual shape of the terrain. For the same reason, the visualization of this data structure is usually stiff in the presentation of terrain details such as ridge lines and valley lines.

In order to show the details of the terrain more realistically and finely, the Triangular Irregular Grid (TIN) was developed. In this type of data, all data points are connected by a set of continuous triangles, which are irregular in size and shape and do not overlap (Yih-ping 1989). In the problem of nonstationary terrain surface visualization, the TIN data structure has a great performance. Because of its irregular nature, its disadvantage is that it needs to consider the rules of data point generation and the calculation method of
processing data points.
The data of terrain model used in this paper is RSG. The main reason for this choice is that this article does not give priority to the accurate visualization of terrain provided by TIN. On the other hand, the goal of this article is the generation of multi-resolution terrain. The RSG data type can greatly simplify the steps and calculations for generating terrain with different resolutions due to the advantages of the regular distribution of data points.

### 2.2 Height Map

The height map is a widely used landscape data set in the RSG data type. Compared with the traditional mesh-based data set, it can correctly represent the information while occupying a smaller storage (Paulo 2008). The height map is essentially an image that represents grayscale, and the value of pixel is between 0 and 255 to represent the grayscale of the pixel. In the use of landscape visualization, the gray values of pixel are regarded as the elevation values at the location of the pixel to record the specific information of the terrain. Due to the simple data structure, height maps become popular as an efficient terrain representation structure. However, each coin has two sides, this data structure is the advantage as well as the disadvantage of the height map. In practice, the real terrain is three-dimensional and complex. One height map with only one elevation value at a location is difficult to describe detailed information such as overhanging obstacles or multi-level terrain. When one location in the landscape has multiple heights, the height map can only select one of the heights (usually the highest one) as the elevation value of the point location. Therefore, the landscape visualized by one height map is actually only a 3D-like scene, which is called 2.5D scene. Nevertheless, the height map is not incapable of visualizing a true 3D scene. In order to use height maps to describe terrain information more accurately, Roberts et al. have used a method of creating multiple height maps for the same terrain to
visualize complex 3D terrain models which including multi-level terrain (Roberts 2008). For this article, an efficient and simple terrain data structure is a suitable choice. A 2.5D scene can meet the requirements. In the next section, we will introduce how the computer processes the map data and visualizes it on the screen.

### 2.3 Rendering

Generally, the dynamic scenes on the computer screen are essentially presented by rapidly updating static pictures. The rate of such displayed images is generally in units of frames per second (fps). In practice, only when the fps of the dynamic picture exceeds 15, the users can feel the sense of interaction and focus on it (Akenine-Möller 2008). In the visualization of terrain, the user can move his position as an observer in the screen and look at the terrain in front of him. the job of computer at this time is to quickly process the received terrain data in real time, and at the same time draw the scene into pictures that the observer should watch according to the observer's position. The area that the observer can see in the scene is called the view frustum, as shown in the figure 1a. The image presented by the computer is the result of the projection of the object, which position between camera and the far clip plane, to the near clip plane in the figure1. Therefore, objects outside the visual vertebral body will not appear on the screen as shown in Figure 1. But these objects have not disappeared, the GPU is still rendering them continuously. Only part of the objects in the visual frustum will be cropped, and only the part inside the view frustum will be displayed on the screen. In addition, when the observer's perspective rotates or moves, the view frustum will rotate or move with it, and the computer will update the picture by calculating the coordinates and the cone.


Figure 1. View Frustum

The procession of computer from receiving data to drawing pictures to the screen is called the graphics rendering pipeline. Tomas et al. divided the rendering pipeline into four stages in Real Time Rendering (Akenine-Möller 2008): application, geometry processing, rasterization, and pixel processing. Based on the terrain visualization in this article, we will introduce the work of the computer in these four stages.

Application: Generally, at this time, all data is in the CPU, and we will process the data to achieve our expected results such as frustum culling and the application of the LOD method. Frustum culling is a way of rendering optimization, which reduces the burden on the GPU by pre-determining the object outside the view frustum and temporarily discarding its data so that the GPU does not need to render the object.


Figure 2. Three Coordinate Systems
Geometry processing: The GPU receives the data which has been processed in the application stage input from the CPU in this stage. After GPU stores the data, it transforms the vertices position data into the world coordinates (The coordinate system with the computer's world origin), the camera coordinate (The coordinate system with the observer position coordinates as the origin) and the image coordinates (the position coordinates of the target vertex in the image) as shown in the figure 2. Finally, after cutting the part outside the frustum, the GPU will get the vertex data needed for picture drawing. Based on these data, the GPU will draw the grid for the landscape.

Rasterization: As shown in Figure 3, the image content presented by the computer is an approximate shape formed by the coloring of individual pixels. In this stage, the computer selects all the pixels to be colored and inputs these data to the next stage.

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |

Figure 3. Rasterization
Pixel coloring: At this time, the computer uses the vertex shading information input from the CPU to the GPU to color the corresponding pixels. It should be noted that the depth test will be performed here at the same time, and the pixel will give priority to the shading information of the vertices that are not occluded.

These steps are the basic process of drawing a picture from the scene. When the computer rendering system is immature, all these steps need to be built from scratch. Fortunately, these rendering steps are similar, and the same rendering framework can be shared to a certain extent. Therefore, a variety of graphic tools gradually appeared in people's vision.

### 2.4 Game Engine and API

At present, developers have created and developed game engines such as Unity, Unreal 4, and application programming interfaces (API) similar to Open Graphics Library (OpenGL) in this field. This not only improves the efficiency of people's development in this field, but also reduces the barriers for people to understand and enter this field. Generally, game engines provide developers with a variety of well-packaged systems such as rendering engines, physics engines, and collision detection. Therefore, developers can focus on developing the game's gameplay and features without having to re-
develop the basic functions required by games.
The tool used in this article is OpenGL. Unlike game engines, OpenGL only provides APIs for part of the rendering pipeline, and other required functions such as lighting systems need to be created and implemented by users. Although it does not provide powerful functions like a game engine, in OpenGL we can build the desired LOD algorithm while creating a rendering engine. On the other hand, using the existing rendering pipeline process in OpenGL, we can focus on the processing of map data and the selection of rendering methods without spending too much time processing data transmission between computer hardware. The main work of this paper focuses on the realization of the LOD method in the application stage as referred. The next section of this article will introduce common LOD algorithms.

### 2.5 The Algorithms for Levels of Detail

The most critical part of the LOD algorithm is the conversion of high and low resolution terrain. The resolution of the terrain is mainly determined by the grid. The number of triangles in the grid determines the resolution of the terrain block and the realism of the displayed effect. Therefore, the processing of triangles in the grid has become the research goal of developers.

Hoppe et al. proposed the Progressive Meshes algorithm to achieve the fusion and splitting between triangle meshes. This algorithm achieves the reduction or grown of the number of vertices and triangles as shown in Figure 4 by traversing the target mesh space and applying the mesh transformations, edge collapse or edge split (Hoppe 1996). Because this method is reversible for the processing of vertices and triangles, it can be used to process the realtime conversion of terrain between different resolutions. Because the Progressive Meshes algorithm model focuses on the microscopic processing of each triangle, it can adapt to the processing of irregular vertex data.

Therefore, this method performs very well when visualizing TIN type map data. On the contrary, when facing RSG-type terrain data, the model conversion of this algorithm is complicated and cumbersome compared to other algorithms, and it will take up a lot of computing resources.


Figure 4. The transformation of edge collapse (Hoppe 1996)
Duchaineau et al. proposed the Real-time Optimally Adapting Meshes (ROAM) algorithm to implement the application of the LOD method (Duchaineau 1997). The algorithm divides the terrain into multiple rightisosceles triangles by constructing a Triangle Bintree, and gradually forms a complete grid space by continuously dividing these triangles. The method of dividing a triangle is to separate the two triangles by connecting the midpoint of the hypotenuse of the right-isosceles triangle and the vertex of the right angle. This method achieves the conversion of different resolution terrains by adjusting the segmentation level. The segmentation process is shown in Figure 5.


Figure 5. Levels 0-5 of a triangle bintree (Duchaineau 1997)

Lindstrom et al. proposed a quadtree method to achieve LOD method (Linstrom 1996). The algorithm constructs a quad-tree, treats the terrain as a rectangle, uses the middle point of the rectangle to divide it into four small rectangles evenly, and repeats this way to finally form a satisfactory terrain grid. This method realizes the conversion of terrain resolution by controlling the segmentation level as well as the ROAM method. The segmentation process is shown in Figure 6.

$\ell=0$

$\ell=1$


$$
\ell=2
$$

Figure 6. Levels 0-2 of a quadtree (Linstrom 1996)
Both the ROAM algorithm and the quadtree algorithm are expert in processing RSG type terrain data. In theory, the ROAM algorithm can more accurately determine the distance between the terrain block and the observer so as to draw a satisfactory grid with fewer triangles. But there are more complicated calculation methods with this accuracy. On the other hand, although the quadtree algorithm has low control accuracy on the grid, it only needs simple calculations to divide the grid. Therefore, this article uses the quadtree algorithm to achieve the goal.

## 3. Visualizing landscapes with a quadtree

### 3.1. Overview over the whole systems

In this section, we will briefly introduce the work done in this article.
In the field of computer graphics, a complete rendering framework is the foundation of all work. Based on the API provided by OpenGL, this article achieves the movement and direction control of the user as an observer in the world. In the same way, the simulated real lighting display based on Phong shading model is realized. These functions, together with the predetermined processing of data such as vertex position coordinates, texture coordinates, and normal vectors, constitute a basic rendering framework. The work done by the rendering framework is mainly focused on the geometric processing and pixel shading stages in the rendering pipeline as mentioned above. Generally, in this framework, all terrain data can be input to the GPU through the API provided by OpenGL to achieve a complete single-resolution terrain visualization. Therefore, in the next step this article will build a quad-tree data structure to store and process all terrain data. At the same time, this article will measure the undulation degree in each terrain block and determine the level of this quadtree branch according to both the value and the distance from the observer. In this process, this article will introduce the elimination method to avoid the appearance of cracks in the rendering process. Finally, before transmitting the processed terrain data, based on the position and direction of the visual frustum, this article will remove the data outside it in advance.

### 3.2 Rendering

### 3.2.1 Vertex Data Transfer

Before explaining the work of this article, we will briefly introduce the interaction between OpenGL and hardware transmission. When we input the vertex data to GPU, usually GPU will open up a part of the memory space for
storing these data. OpenGL provides us with an API - Vertex Buffer Object (VBO) that allows us to manage this data. Since the vertex data contains different vertex attributes such as position, texture, normal, etc., in order to reduce the re-retrieval and recall of the VBO when the GPU uses different attributes, we need to create a Vertex Array Object (VAO) to bind the current VBO information to simplify the communication between GPU and VBO. After creating the VBO and VAO, we can tell GPU the index value of the three vertices of each triangle in the triangle mesh to be drawn, so that GPU can draw triangles more efficiently. These index data we input is called Index Buffer Object (IBO) or Element Buffer Object (EBO). Under the synergy of these three objects, GPU can successfully visualize the vertex data to the screen. Each time the computer passes through this process, GPU can render an object. However, when the object has a huge amount of vertex data, GPU may not be able to open up a large enough storage space, causing rendering failure. It just happens that the quadtree algorithm of this article can avoid this situation.


Figure 7. The crack between the blocks of leaf nodes
This article binds all the vertices data to the leaf nodes of the quadtree. During rendering, all leaf nodes transfer their own data one by one into GPU. Therefore, the overall terrain visualized in this article is spliced by the squares formed by each leaf node. Accordingly, if these leaf nodes are not spliced well,
the situation shown in the figure 7 will appear. We can clearly observe the edges of each terrain block, these terrain blocks are not well spliced into a whole landscape. This article will introduce how to avoid this situation by processing the data in EBO in the quadtree subsection.

### 3.2.2 Camera

The camera is what we call the observer, and the scene it sees is the pictures presented to us by the computer after calculation. In fact, there is no concept of a camera in OpenGL, or in other words, it only provides us with a static camera. We can only make us believe that the camera is moving by moving all the objects in the scene. Therefore, it should be noted that the direction the objects move in the calculation is opposite to the direction perceived by the camera.


Figure 8. The attributes of the camera
As shown in Figure 8, the camera has four basic attributes: position, facing direction vector, right vector and up vector. Using these four basic properties, we can get an observation coordinate system with the camera as the origin. Through the $x, y$, and $z$ axes of the observation coordinate system, which are the three vector attributes, we can get a matrix that transforms the world coordinates of any object into observation coordinates. This matrix is called the LookAt matrix:

$$
\text { LookAt }=\left[\begin{array}{cccc}
R_{x} & R_{y} & R_{z} & 0 \\
U_{x} & U_{y} & U_{z} & 0 \\
D_{x} & D_{y} & D_{z} & 0 \\
0 & 0 & 0 & 1
\end{array}\right]\left[\begin{array}{cccc}
1 & 0 & 0 & -P_{x} \\
0 & 1 & 0 & -P_{y} \\
0 & 0 & 1 & -P_{z} \\
0 & 0 & 0 & 1
\end{array}\right]
$$

Where $R$ is the right vector, $U$ is the up vector, $D$ is the direction vector,
and $P$ is the camera position vector. Because we need the world to move in the opposite direction, the P vector is negative.

Our first step is to achieve the control of camera displacement. Set the vector F as the camera's forward direction. After obtaining the corresponding keys from the keyboard, set the camera's movement speed to vor -v (set the speed to 0 when no input is detected). Multiply the current speed v by the F, R and $U$ vectors in real time. Finally, the result obtained is added to the position of the camera to control the movement of the camera. The formula is as follows. This article assumes that 'W' is forward, 'S' is backward, 'A' is left, 'D' is right, ' $Q$ ' is upward, and ' $E$ ' is downward.

$$
\text { Position }+=F * v_{F}+R * v_{R}+U * v_{u}
$$

The second step is to implement the control the visual frustum. The rotation of the visual frustum is equivalent to the head-turning action of a character in the real world. We use Euler angle as the recording method of rotation angle, as shown in the figure 9 . Generally, for a camera, the use of pitch and yaw is sufficient to record the rotation angle of the lens.


Figure 9. The Euler angles
We get the displacement of the mouse in real time to calculate the realtime pitch and yaw values. Finally, the direction the camera faces is calculated by the following formula:

$$
\begin{aligned}
& \text { Direction. } x=\cos (\text { pitch }) * \cos (\text { yaw }) \\
& \text { Direction. } y=\sin (\text { pitch }) \\
& \text { Direction. } z=\cos (\text { pitch }) * \sin (y a w)
\end{aligned}
$$

### 3.2.3 Illumination

In practice, lighting is a virtual light source created by a computer. The light source can be invisible, and the computer will simulate the influence of the light source on the brightness of the object surface. The implementation method is to mix or multiply the color vector of the vertex texture with the color vector of the light source. There are already many ideal lighting models to choose from. This article will briefly introduce two lighting models: Lambert and Phong Shading.

The lighting phenomenon simulated by the Lambert model is diffuse reflection. Diffuse reflection is mainly formed by the light reflected on the rough surface of the object from the light source. The final brightness is related to the angle between the incident light and the normal of the point, as shown in Figure 10. The calculation method of the model light is:

$$
L_{d}=C * \max (0, \cos (\hat{l}, \hat{n}))
$$

Where $\hat{l}$ represents the incident light, $\hat{n}$ represents the normal vector of the point, $C$ represents the intensity and color of the light.


Figure 10. Diffuse reflection

Phong Shading model adds a specular reflection to the Lambert model. This model allows the observer to see the light source on the surface of the object, just like the reflection of a mirror. Therefore, the brightness of the specular reflection is affected by the observer's position. As shown in the figure 11, the brightness and the reflected light $R$ are related to the angle of
the observer's line of sight.


Figure 11. Specular reflection
The calculation formula is:

$$
L_{\text {specular }}=C * \operatorname{pow}(\max (0, \cos (r, v)), p)
$$

Where $r$ represents the reflected light, v represents the line of sight, C represents the intensity and color of the light, p is a constant.

Since the target of visualization in this article is landscape, its surface is rough and generally does not appear specular reflection. Therefore, this article only uses the simulation of diffuse light and ambient light in Phong Shading model. Ambient light generally refers to the brightness caused by the reflected light without light source. For instance, a shadow that is not directly illuminated by a light source in real life but can still have a certain brightness is caused by ambient light. The calculation formula is:

$$
\text { ambColor }=\text { ambStrength } * \text { lightColor } * \text { objectColor }
$$

Where, ambStrength is the ambient light constant, generally set to 0.1; lightColor is the ambient light color; objectColor is the color of the object at that point.

For the calculation of diffuse reflection, the angle $\theta$ between the light direction and the normal vector shown in the figure 10 can be obtained by the dot product of the incident light direction vector and the normal vector. The final calculation formula for diffuse reflection is:

$$
\text { diffColor }=\max (\operatorname{dot}(l, n), 0) * \text { lightColor }
$$

Finally, in the shader, the result of adding the brightness of the two lights is multiplied by the model color which obtained from the texture data of the position:

$$
\text { resultColor }=(\text { diffColor }+ \text { ambColor }) * \text { objectColor }
$$

### 3.3 Quadtree data structure

Quadtree is proposed to solve the problem of the storage and access for discrete data. The earliest quadtree was a point quadtree invented by Finkel and Bentley (Samet 2006). This quadtree is formed by building a twodimensional binary search tree. It saves all point data in the node. As shown in Figure 12, take the appropriate point data position as the root node, select the appropriate point data in the four quadrants as the child nodes, and repeat this step until all the point data have corresponding nodes. Eventually a complete quadtree is formed.


Figure 12. Application and structure of point quadtree

Unfortunately, using a point quadtree to process terrain data is inappropriate. We found a structure called Matrix Quad-prefix Tree (Samet 2006). As shown in Figure 13, this kind of quadtree stores all data information only in the leaf nodes. The selection of the center point of the node is regular, and the position in the middle is selected as the center of the next node. In the MX quadtree, each leaf node corresponds to a 1*1 matrix and contains a point
data.


Figure 13. Application and structure of MX quadtree

We found that these two quadtree structures are applied to discrete data. Therefore, to apply the quadtree to terrain data, we need to adjust the structure of the MX quadtree. In each leaf node, we need it to contain 9 data points including the center point. As shown in Figure 14 and 15, each leaf node can contain point data information that can form a rectangle.


Figure 14. The rectangle corresponding to the quadtree and the structure


Figure 15. Vertices data of a quadtree node

Through the application of the vertex index data, which we mentioned above, the index data of the vertices can be stored in the leaf nodes. Finally, by adjusting the segmentation level of the leaf nodes, we can control the resolution of the terrain block.

In this way, we can achieve the goal of this article: use a quadtree to achieve an adaptive real-time variable resolution grid to optimize the occupation of computing resources for terrain rendering.

The flow chart for constructing a quadtree in this article as shown in Figure 16:


Figure 16. The flow chart for constructing a quadtree

### 3.4 Refinements

This article will introduce the completed work in the sequence of the flowchart.

### 3.4.1 Generation and storage of terrain data

Due to the rules of quadtree segmentation, we need to limit the resolution of the height map. The aspect ratio of the height map must be $1: 1$, and its resolution must meet the requirements of $\left(2^{n}+1\right)$ * $\left(2^{n}+1\right)$. The height map that cannot meet the requirements will be abnormal in the position of the node when it is split. The resolution of the height map used in this article is 1025*1025, which can meet the restriction requirements.

The terrain vertices data in this paper has three attributes: position, texture and normal vector. These three attributes correspond to the coordinates of the vertices, shading data, and lighting data.

Vertex position generation and storage: This article uses a twodimensional array to store the data which obtained from the height map. The gray value of each pixel in the height map is stored in the corresponding array position. Based on this array, we will generate the corresponding coordinates for each pixel, in other words, we will supplement them with the $x$-axis and $z$ axis coordinates. Finally, we use VBO to input these coordinate values into GPU.

Generation and storage of texture attributes: The texture data is similar to the height map data and needs to be obtained from a texture picture. The difference is that the texture data is a two-dimensional coordinate value, and its corresponding position is the coordinate of the pixel on the texture image. In addition, each pixel of the texture image has three values, corresponding to the Red, Green and Blue (RGB) values. Based on the coordinates in the texture data, the GPU inputs the corresponding RGB values on the texture image to the rendering pipeline for pixel shading. The values of texture coordinates are as shown in the figure 17. Generally, their value ranges are
between $0-1$, and the coordinates of the out-of-range range can be taken by splicing the same pictures.


Figure 17. Texture coordinate system
Normal vector generation and storage: Due to the importance of vertex normal to the lighting model, researchers have developed many methods for calculating vertex normal. The accuracy of vertex normal has a great influence on the realism of lighting. However, this article does not have high requirements for the lighting model. Therefore, the most basic vertex normal calculation method is selected in this article. We refer to the Mean Weight Equally (MWE) algorithm introduced by Henri Gouraud (Jin 2005). The algorithm takes the average of the normal vectors of the patches around the vertex to get the normal of the point. In this paper, the normal vectors of the surrounding four patches are obtained by calculating the cross product of the vectors between the vertex and the points of up, down, left and right of the vertex. We average these four normal vectors to get the result. Finally, all the vertex normal data are input to the GPU to wait for the use of the lighting model.

### 3.4.2 Elimination of objects outside the view frustum

We have briefly introduced the frustum and its imaging principles in the background chapter. At the same time, we can obtain the real-time
coordinates of the object in the observation coordinate system by multiplying the world coordinates of the object with the LookAt matrix of the camera. In the observation coordinate system, we can calculate the normals of the six faces of the view frustum. Through the product of the coordinates of the object in the observation coordinate system and the normals of the six faces of the frustum, we can determine whether the object is in the frustum.

However, the objects have their own volume. We need to calculate the outline of the object in case it is inside the frustum but not rendered. We need to calculate the most extreme coordinates of the object in six different directions. Each coordinate will be judged in turn, and we will reject the object when all the extreme positions of the object are outside the frustum.

### 3.4.3 Segmentation of Quadtree Nodes

This section will introduce the conditions used in this article to determine whether a quadtree node can be divided. Below we will call it the node evaluation system. Generally, the node evaluation system is composed of multiple factors. The most important factor is the distance between the terrain block and the observer. Therefore, we need to calculate this distance $l$ through a formula. In fact, we can get the result with the commonly used Euclidean distance:

$$
l=\sqrt{\left(c_{x}-b_{x}\right)^{2}+\left(c_{y}-b_{y}\right)^{2}+\left(c_{z}-b_{z}\right)^{2}}
$$

Where $\left(c_{x}, c_{y}, c_{z}\right)$ represents the location of the camera, $\left(b_{x}, b_{y}, b_{z}\right)$ represents the location of the center point of the terrain block.

For the node evaluation system, only one element of distance is not a good representation of the obvious features of the terrain. Stefan et al. proposed a node evaluation system with three factors (Röttger 1998). Of these three factors, one is the distance $l$, one is the side length $d$ of the terrain block, and the other is the slope change proposed by Peter Linstrom et al. (Linstrom 1996), which can also be called roughness. Distance and side length are relatively intuitive data. We will briefly introduce the principle of
slope change and its calculation method.
Since the change of the vertices in a terrain block is irregular, we need to adapt to the slope change to indicate the degree of change in the height of the vertices in the terrain block. As shown in Figure 18, $\delta_{B}, \delta_{F}$ and $\delta_{D}$ are the expressions of slope changes. The calculation method is

$$
\delta_{B}=\left|B_{z}-\frac{A_{z}+C_{z}}{2}\right|
$$

The calculation method of $\delta_{F}$ and $\delta_{D}$ is similar to that of $\delta_{B}$. Generally, a terrain block can calculate four slope changes, and we generally choose the largest value among them as the value for the slope change of the terrain block:

$$
R=\max \left(\delta_{B}, \delta_{F}, \delta_{H}, \delta_{D}\right)
$$



Figure 18. Roughness of the block (Linstrom 1996)
In the node evaluation system proposed by Stefan et al., they combined these three factors into the following formula:

$$
f=\frac{l}{R \times d}<C
$$

Where $C$ is a constant, as the threshold for segmentation. When the value of $f$ is less than $C$, the node is split, otherwise it will not be split and input to the rendering queue.

This article adjusts this method, removes the factor of terrain block side length $d$, and uses only terrain roughness and distance as the node evaluation system. The formula is:

$$
f=\frac{l}{R}<C
$$

On the other hand, due to the crack elimination method in this article, besides these two factors, there is another factor that needs to be taken into consideration. Because this factor is independent of this node evaluation system, we will introduce it in detail in the next section.

### 3.4.4 Crack elimination

In the above, we mentioned that the terrain drawn in this article is composed of many small rectangular blocks. Therefore, we need to pay special attention to the treatment of the common edges of adjacent rectangular blocks. After setting the quadtree node evaluation system, we will render terrain with different resolutions. Cracks will appear at the junction of terrain blocks of different resolutions. The reason for the cracks is that different nodes have different heights at the same coordinate position. As shown in Figure 19, the two sides of the crack are nodes with different division levels. The node A with a higher division level will have more vertices on the same edge than the other node B. At this point, if the height of the extra vertex is not consistent with the height of the edge rendered by node $B$, cracks will appear.


Figure 19. The appearance of cracks
There are many ways to deal with cracks. For example, the filling method (Wu 2010) is to render a plane at the crack with the point where the crack is
generated to fill the crack. This method needs to locate the position of the crack after constructing all the leaf nodes of the quadtree, and then render the plane to fill the crack.

This article uses the elimination method. Compared with the filling method, this method has the advantage that the crack can be eliminated without locating the position of the crack (Linstrom 2002). Through the above content, we know how the cracks are produced, and the elimination method directly eliminates the causes. By judging the division levels of two adjacent nodes, the extra vertices in the high-resolution node are removed during rendering, as shown in Figure 20. The number of vertices on this adjacent side remains unchanged, thus eliminating the occurrence of cracks.


Figure 20. Crack elimination
The only limitation of this method is that the division level of adjacent nodes cannot be greater than 1. Once this limit is exceeded, a situation as shown in Figure 21 will appear. It is impossible to eliminate nodes only by eliminating the nodes in the middle. Therefore, we need to add an array to indicate the division of each node.


Figure 21. Unmanageable crack
In order to implement this method, we need to implement a way to record the division status of adjacent nodes. There is a condition that needs attention: when we split a node, we need to judge the split state of neighboring nodes. If the depth-first algorithm is used for segmentation, the priority of adjacent node segmentation may be lower than that of the child nodes of the current node, causing the child node to fail to detect the children of the adjacent node. Therefore, we use the breadth first algorithm when segmenting. Next, we create a two-dimensional array to record the segmentation status of each node, where the recorded data represents the segmentation status of the node centered on the current position. In the process of recording the segmentation status of the array, we found that we only need to record whether the current node will continue to be segmented. The reason is that once a node does not continue to split, neighboring nodes around it will also stop splitting. Therefore, when recording the array, it is only necessary to determine whether the target node will continue to be split, if the split is no longer continued, then its four child nodes will be recorded as 0 ; otherwise, it will be recorded as 1. As shown in Figure 22


Figure 22. Assignment of flag array

### 3.4.5 Grid visualization

OpenGL provides developers with three methods as shown in Figure C when drawing triangles: GL_TRIANGLES, GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN. The first method is to draw the vertex data transmitted into the GPU in groups of three. As shown in figure 22.a, when we input 6 vertex indexes, the computer will draw 2 triangles based on these six vertices. This method is the most basic and most used drawing method. The STRIP method draws the first triangle from the vertices $V_{0}, V_{1}$, and $V_{2}$ shown in Figure 22.b. When processing $\mathrm{V}_{3}$, the $\mathrm{V}_{0}$ will be discarded, and the second triangle is drawn with $\mathrm{V}_{1}, \mathrm{~V}_{2}$, and $\mathrm{V}_{3}$ as the vertices, and so on. Through the description, we find that this method improves the usage rate of index of vertex data. As shown in Figure 22.b, four triangles are drawn when only 6 vertex indexes are input. This method can greatly reduce the input vertex index data when drawing special graphics. The FAN method will draw a triangle as shown in Figure 22.c. Take the first input data $V_{0}$ as the fixed vertex and draw the first triangle after receiving $\mathrm{V}_{1}$ and $\mathrm{V}_{2}$. When $\mathrm{V}_{3}$ appears, it will discard $\mathrm{V}_{1}$ and draw the next triangle with $\mathrm{V}_{0}, \mathrm{~V}_{2}, \mathrm{~V}_{3}$ as the vertices, and so on. Finally, a fan-shaped grid is drawn. Compared with the first method, the number of input vertex indexes can also be reduced.


Figure 23. Three methods to draw triangles
In addition, because of the need to draw each leaf node of the quadtree in this article, the FAN method performs very well for the rendered terrain blocks. As shown in Figure 23, when rendering each node, we only need to enter 10 vertex indexes to render the 8 triangles required by the node. If we render such a terrain block grid in the first way, we need to input 24 vertex indexes.


Figure 23. Vertex input order when rendering terrain blocks
After choosing the rendering method, we also need to deal with the cracks when rendering. In the above, we have recorded the segmentation status of all nodes through a two-dimensional array. When rendering a node, we need to use this data to determine the segmentation state of surrounding nodes, as
shown in Figure 24. The current drawing node detects that the record value of the right node is 0 , and the input vertex index will ignore the corresponding edge vertex - the 5th vertex, as shown in Figure 24.a and 24.b. Finally, the grid shown in Figure 24.c can be drawn, which successfully eliminates the effect of cracks.


Figure 24. Crack elimination

## 4. Results and Analysis.

### 4.1 Crack

In the process of cracks, we can find that the crack elimination method used in this article solves the problem of cracks by comparing the pictures (a) and (b) of Figure 25.


Figure 25. The result before and after the crack elimination
We can observe the results of crack treatment through the display of the grid. In the figure 26, we can see four consecutive levels of quadtree leaf nodes and visually see the results of each node's drawing. In the four levels of nodes from right to left, none of the vertices that will cause cracks on the right are drawn. In the node with the highest segmentation level, we can also see the appearance of nodes with more vertices that have not been rendered.


Figure 26. The polygon mode after removing cracks

On the contrary, the crack elimination algorithm in this paper has certain flaw. The luminous and darker wireframes or scene in the Figure 27 are affected by the light in this article. We can find that there is no occlusion of light (there is no height change) in some of the luminous places and dark places. This is because the vertex normal data in this article is calculated based on the full-resolution height map data. When the segmentation level of the node is not enough to render the feature of terrain in it, the lighting display will be wrong. Besides the problem of the lighting model, there is also the problem of crack elimination algorithms. Since the gap of splitting level of adjacent nodes cannot be more than one level, the nodes that should be rendered with significant feature cannot continue to be split. In the end, it was inevitable that this situation occurred.


Figure 27. Abnormal light in the scene

### 4.2 Display of rendering results

This article mainly uses height map data with a resolution of 1025*1025. After applying the quadtree algorithm of this article, the final grid drawing result is shown in Figure 28. We can find that the terrain block closer to the camera is drawn with a smaller terrain grid, and the terrain block farther from the camera is drawn with a larger terrain grid. Similarly, the rugged terrain also uses a higher resolution grid.


Figure 28. The polygon mode of the landscape
In addition, according to the terrain effect shown in the figure 29, we find that the edges of the terrain in the vicinity are smoother than the edges of the terrain far away from the camera. This result achieved the goal of this article.


Figure 29. The result of the terrain visualization

### 4.3 Simplification efficiency of terrain grid

In addition, we need to calculate the simplification efficiency of the quadtree structure of this article on the terrain grid. In order to show the simplified efficiency of the drawn multi-resolution terrain model more intuitively, this paper tests the optimization of the number of triangles drawn based on height maps of different resolutions in table 1.

| The <br> resolution <br> of height <br> map | Without <br> quadtree | With quadtree but <br> Without clip |  | With quadtree and clip <br> Triangles | Number of <br> Triangles |
| :--- | :--- | :--- | :--- | :--- | :--- |
| Rendering <br> rate | Number of <br> Triangles | Rendering <br> rate |  |  |  |
| $1025 *$ <br> 1025 | 2094288 | 10216 | $1.95 \%$ | 8922 | $1.70 \%$ |
| 2049 <br> 2049 | 8388608 | 52637 | $0.63 \%$ | 26442 | $0.32 \%$ |

Table 1. The simplified efficiency of quadtree
Since this article is rendering a real-time variable resolution terrain grid, the above data will change with the location of the camera and the ruggedness of the terrain data of the selected height map. Even so, the data still has considerable reference value. We can find that after using the quadtree algorithm in this article, the number of triangles during rendering is greatly reduced, and the rendering efficiency of the computer is greatly optimized.

### 4.4 Compare the terrain with full details

This section will compare the variable multi-resolution model rendered in this article with the highest resolution single model of the original height map. In order to control the influence of the number of triangles in the field of view on the rendering frame rate, this article chooses the same scene as much as possible when comparing.

In the following four sets of pictures, (a) are all pictures rendered using the highest resolution model. The pictures (b) are rendered in the same position through the variable multi-resolution model established in this article. The pictures (c) are rendered after adjusting the segmentation threshold based on the pictures (b). The number on the left of each picture records the FPS at the current moment.


Figure 30. Results of different resolutions and thresholds

(a)
(b)

(c)

Figure 31. Grid results with different resolutions and thresholds


Figure 32. Results of different resolutions and thresholds

(a)

(b)

(c)

Figure 33. Grid results of different resolutions and thresholds

We can intuitively feel that the terrain shown in (a) are real and detailed, but the frame rate in Figures above is only 2 frames per second. This is an unacceptable visual effect for users since there is no sense of interaction.

On the other hand, the difference between (b) and (a) is more obvious where the terrain roughness is smaller, and there are many details that are not rendered which shown in Figure 30 and 31. Even though, the performance effect of (b) in Figures 32 and 33 is slightly different from that in Figures (a). In addition, the frame rate of pictures (b) in the four groups of pictures can be maintained above 30 frames per second, which greatly optimizes the rendering efficiency of the GPU.

Finally, if we want to improve the accuracy of rendering, we can adjust the threshold of node segmentation to get the result shown in Figures (c). We can find that there is a significant improvement in places with less roughness, but the frame rate is reduced.

### 4.5 Compare different scenes

The terrain displayed in this article has different performances in different scenes. Since this article uses the function of removing nodes outside the frustum, the frame rate will fluctuate to a certain extent when the observer is looking at different scenes.

Figures (a) and (b) show insignificant frame rate changes. But in Figure (c). When the observer looks at the nearby terrain block, the frame rate increases from 35 to 103 . This shows that the frustum culling method in this paper has successfully improved the rendering speed. However, it does not always increase the frame rate by a large margin when looking at the near terrain. The frame rate of the picture shown in Figure (d) has only increased to 41. This is due to the imperfection of the frustum algorithm in this paper. In general, the frustum algorithm also needs to eliminate the rendering of occluded objects, but this article does not implement it.


Figure 34. Results of different scenes

## 5. Conclusions and future work

Through the display of the results of the previous section, this article basically solves the problems raised in the first chapter. Based on greatly reducing the number of rendered triangles, the quality of the visualized terrain is guaranteed. Established a real-time variable resolution terrain model and smoothed the connection of its different resolutions to eliminate the cracks and realized the basic LOD model through the quad-tree algorithm.

Although the algorithm used in this article has achieved the goal, there are still many areas that need to be optimized and improved.

Terrain feature detection and display: In the result figure of the previous section, we found that there are still many obvious features of the terrain will only be displayed when the camera is close enough, which will give the observer a sense of abruptness. At the same time, when observing
unrendered features from a distance, the light reflection at that position will be abnormal, that is, shadows will appear without obstructions. For this improvement, this article believes that it is necessary to improve the record array for terrain ruggedness in this article and use more accurate algorithms to evaluate and record the data. More than that, the factors and calculations of the node evaluation system need to be improved as well.

The utilization efficiency of GPU and CPU: In the process of testing in this article, we found that the GPU and CPU occupancy is low, but the displayed frame rate can only reach about 100 frames per second or even lower. Therefore, the rendering pipeline used in this article needs to be improved in the future, so that it can make full use of the capabilities of the GPU and CPU to achieve higher frame rates.

The realism of the scene: This article only considers the texture map and lighting of the terrain, and the details of the terrain are not satisfactory. In this regard, the application of tessellation and more different types of texture maps can be considered.

Normal calculation: All the vertices normals in this article are calculated based on the highest resolution vertex data. Therefore, in the rendered scene, there will be lighting effects that do not match the terrain. In the future, if we want to improve this aspect, we need to recalculate the vertex normal for each leaf node. We can get the lighting effect that matches the actual rendered terrain.

Optimization of crack elimination algorithm: Although the crack elimination algorithm in this article does not need to find the location of each crack, its limitation still makes the rendering effect unsatisfactory. In the future, we can try to use the crack filling algorithm to seek further optimization.

## 6. Reflection

In this project, I was exposed to the field of terrain visualization for the first time. At the beginning of the project, the first thing I faced was the choice of rendering platform or API. I chose Unity for the first time to start research, because it is more complete and easier to use, and more importantly, it is more attractive to me. But after I have studied the basic operations and content of Unity in depth, I found that it is more suitable for research on projects like game network structure or collision detection. Because it already has a complete rendering engine, you need to have a deep understanding of the structure of the Unity rendering engine if you want to access the vertex data in the terrain.

In the end, I chose OpenGL based rendering engine as the starting line. Fortunately, I have learned the basics of OpenGL and computer graphics in the course. But in actual operation, I found that I did not really understand this knowledge. For example, I know the calculation formulas for the six planes of the view frustum, but I don't know how to use these formulas to exclude objects outside the frustum. I achieved it only after reading various articles. Therefore, I understand that the theory in the computer field is only the basis for ensuring the success of practice. The real result is obtained after countless executions and adjustments. Just switching from rendering a complete terrain to using a quadtree to render the terrain, I read various articles and experienced countless times of practice to succeed.

When building the quadtree algorithm, I learned how to use the breakpoint debugging in Visual Studio more efficiently. This is very beneficial to me, because this function is also a skill that must be mastered in most programming language environments. When I only used the quadtree structure to successfully render the terrain (without using the LOD algorithm), I was already thanking Goddess of Luck for her favor.

For the implementation of the LOD algorithm, I only used distance as the criterion for node division at the beginning. However, apart from the cracks, the disappearance of the distant landscape also made me unacceptable. At this time, we need to thank the researchers for their selfless dedication, providing a large number of papers and studies to provide people with multiple choices when encountering difficulties. I am very fortunate to use roughness as another criterion for node segmentation. Because the use of roughness provides great convenience for the crack elimination algorithm I choose. Due to my crack elimination algorithm, the division of nodes will be restricted by neighboring nodes. If there is no roughness as the division criterion, the terrain I finally draw will become a plane composed of only a few nodes (the largest node will limit the further division of other nodes). At this stage, I also understood the advantages of common algorithms, such as recursive algorithms, the use of queues, depth-first and breadth-first algorithms, etc. If I do not have this knowledge, it will be difficult for me to achieve my goal.

The last and most important thing is the ability to read literature. When I am going to do a new project, I will not read the contents of the literature in depth at the beginning. I will extract the basic content that I should know. After understanding the principles and mastering enough practical experience, I will go to a deeper understanding of the problems described in the article. When I read the literature on quadtrees at the beginning, I didn't even know what the cracks were. After I actually practiced and cracks appeared, I understood the meaning of cracks described in the literature.

## References

Mark de B. and Kartrin, T. G. 1998. On Levels of Detail in Terrain. Department of Computer Science 60(1), pp. 1-12.

Emanuele D, Leila D.F, Laura P and Maria V. 2006. A Multi-resolution Representation for Terrain Morphology. In: Raubal M., Miller H.J., Frank A.U., Goodchild M.F. (eds) Geographic Information Science. GIScience 2006. Lecture Notes in Computer Science, vol 4197. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11863939 3

Yih-ping H. 1989. Triangular irregular network generation and topographical modeling. Computers in Industry 12(3), pp. 203-213

Paulo Santos, Rodrigo de Toledo, and Marcelo Gattass. 2008. Solid height-map sets: modeling and visualization. Proceedings of the 2008 ACM symposium on Solid and physical modeling (SPM '08). Association for Computing Machinery, New York, NY, USA, 359-365. DOI:https://doi.org/10.1145/1364901.1364953
G. Roberts, S. Balakirsky, and S. Foufou. 2008. 3D Reconstruction of rough terrain for USARSim using a height-map method. Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems (PerMIS '08). Association for Computing Machinery, New York, NY, USA, 259-264. DOI:https://doi.org/10.1145/1774674.1774715

Akenine-Möller, T., Haines, E., \& Hoffman, N. 2008. Real-Time Rendering 3rd ed. A K Peters/CRC Press. https://doi.org/10.1201/9781315365459

Hoppe, H. 1996. Progressive meshes. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, pp. 99-108.

Duchaineau, M., Wolinsky, M., Sigeti, D. E., Miller, M. C., Aldrich, C., \& MineevWeinstein, M. B. 1997. ROAMing terrain: Real-time optimally adapting meshes. In

Proceedings. Visualization'97 (Cat. No. 97CB36155) (pp. 81-88). IEEE.

Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., \& Turner, G. A. 1996. Real-time, continuous level of detail rendering of height fields. In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, pp. 109118.

Samet, H. 2006. Foundations of multidimensional and metric data structures. Morgan Kaufmann.

Jin, S., Lewis, R. R., \& West, D. 2005. A comparison of algorithms for vertex normal computation. The visual computer, 21(1), pp. 71-82.

Röttger, S., Heidrich, W., Slusallek, P., \& Seidel, H. P. 1998. Real-time generation of continuous levels of detail for height fields. University Erlangen-Nürnberg

Wu, J., Yang, Y., Gong, S. R., \& Cui, Z. 2010. A New Quadtree-based Terrain LOD Algorithm. J. Softw., 5(7), pp. 769-776.
P. Lindstrom and V. Pascucci. 2002. Terrain simplification simplified: a general framework for view-dependent out-of-core visualization. IEEE Transactions on Visualization and Computer Graphics, 8(3), pp. 239-254, doi: 10.1109/TVCG.2002.1021577.

