Managing and rendering large environments in games

Hazrat Pradipta Ranjali

2010

Hazrat, P. R. (2010). Managing and rendering large environments in games. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/36281>

<https://doi.org/10.32657/10356/36281>

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

MANAGING AND RENDERING
LARGE ENVIRONMENTS IN GAMES

HAZRAT PRADIPTA RANJALI

MANAGING AND RENDERING LARGE ENVIRONMENTS IN GAMES

2010

**HAZRAT PRADIPTA RANJALI
SCHOOL OF COMPUTER ENGINEERING
2010**

MANAGING AND RENDERING LARGE ENVIRONMENTS IN GAMES

HAZRAT PRADIPTA RANJALI

SCHOOL OF COMPUTER ENGINEERING

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Master of Engineering

2010

Acknowledgements

The author would like to express his most sincere gratitude towards his project supervisors, Professor Seah Hock Soon, Parimal Aswani, and Assistant Professor Tian Feng who have given him invaluable advice and guidance during the term of his project.

The author would also like to thank the staff in TQ Global Pte. Ltd. for providing him the chance to work on this project, excellent work environment and all the resources needed.

Lastly, the author would like to thank his friends for their encouragements and suggestions which allow him to solve the problems that he faced during the development of the project.

Contents

Acknowledgements	iii
Contents.....	iv
List of Tables	vi
List of Figures.....	vii
Abstract.....	ix
Chapter 1 Introduction.....	1
1.1 Motivation	1
1.2 Scope and Objectives	5
1.3 Overview.....	7
Chapter 2 Background.....	9
2.1 Terrain Data Structure	9
2.2 LOD Rendering Algorithms	14
2.2.1 Continuous LOD for Height Fields	17
2.2.2 ROAM	21
2.2.3 Geometrical MipMapping	23
2.2.4 Geometry Clipmaps	29
2.3 GPU Programming	33
2.4 Scene Graph	38
Chapter 3 Terrain Implementation.....	40
3.1 General Problem	40
3.2 BinTrees Hierarchy.....	43
3.3 Tiled Block	47
3.3.1 Data Structure	48
3.3.2 Rendering.....	50
3.3.3 Terrain Collision	53
3.4 Nested Regular Grids	54
3.4.1 Data Structure	55
3.4.2 Rendering.....	57
3.4.3 Terrain Collision	57
3.5 GPU Optimizations	58
3.5.1 Branching	59
3.5.2 Instruction-level parallelism.....	60
3.5.3 Precomputation of loop invariants.....	60

Chapter 4	Experiment and Analysis.....	62
4.1	Algorithms Comparison	62
4.2	Proposed Implementation	66
4.2.1	Data Streaming	67
4.2.2	Stitching	68
4.2.3	Collision Time.....	71
4.3	Terrain Collision Detection Optimization	73
Chapter 5	Summary	77
5.1	Summary	77
5.2	Contribution	78
5.2.1	Data Streaming	78
5.2.2	Improved Stitching	78
5.2.3	GPU Terrain Collision	79
5.3	Future work.....	80
Chapter 6	Appendix	81
6.1	Visual Artefact in GeoMipMap.....	81
6.1.1	First Example	81
6.1.1	Second Example	85
6.1.2	Third Example	88
References		91

List of Tables

Table 2.1	Comparison between TIN and heightmap[14].....	10
Table 4.1	Comparison of terrain rendering algorithms.....	62
Table 4.2	Data streaming time comparison.....	67
Table 4.3	Stitching comparison.....	68
Table 4.4	Collision time comparison.....	71
Table 4.5	Collision time comparison using object categorization.....	75
Table 4.6	Collision time comparison using time step.....	76

List of Figures

Figure 1.1	Terrain rendering on Battlefield games[2]	2
Figure 2.1	TIN (left) and Regular grid (right) data structure[13]	10
Figure 2.2	Input and output of TIN [18].....	12
Figure 2.3	Heightmap and output	13
Figure 2.4	Geometric representation of delta values[5]	18
Figure 2.5	Obtaining lower resolution blocks	20
Figure 2.6	T-Junction and crack	21
Figure 2.7	Triangle Bin-Tree level	22
Figure 2.8	Split-Merge operation	22
Figure 2.9	Texture mipmapping.....	24
Figure 2.10	Adjacent blocks of different level of details	27
Figure 2.11	Solving T-junction by stitching	28
Figure 2.12	Rings of texture resolution [40].....	29
Figure 2.13	Geoclipmap regions (left) and terrain (right)	30
Figure 2.14	Toroidal addressing	31
Figure 2.15	Zero area triangles (red).....	32
Figure 2.16	Direct3D Graphics Pipeline[41].....	33
Figure 3.1	BinTree Hierarchy in ROAM.....	43
Figure 3.2	Hierarchy of collision data for ROAM.....	45
Figure 3.3	Triangle relationship in ROAM[53].....	46
Figure 3.4	Tiled Block in GeoMipMaps	47
Figure 3.5	Active area on terrain block	49
Figure 3.6	Visual artefact (red) due to wrong stitching.....	51
Figure 3.7	Stitching proposed by Larsen[1]	52
Figure 3.8	Stitching proposed by Brodersen[37].....	52
Figure 3.9	Visual artefact (red) due to wrong stitching.....	53
Figure 3.10	Nested Regular Grid in GeoClipMap	54
Figure 3.11	Blocks in clipmap level[9]	55
Figure 3.12	Texture merging process.....	56
Figure 3.13	1D arrays packed into 2D texture	58
Figure 4.1	Initial rendering of ROAM in wireframe mode	64
Figure 4.2	Initial rendering of ROAM in wireframe mode at another instant.....	64

Figure 4.3	Visual artefact in GeoMipMap.....	69
Figure 4.4	Original stitching in GeoMipMap.....	69
Figure 4.5	No visual artefact in new implementation.....	70
Figure 4.6	New stitching.....	71
Figure 4.7	Terrain collision algorithm for tiled block terrain.....	72
Figure 4.8	Collision time comparison of CPU and GPU implementations.....	72
Figure 6.1	Original Rendering	82
Figure 6.2	Original stitching.....	82
Figure 6.3	Proposed rendering by Larsen	83
Figure 6.4	Proposed stitching by Larsen	83
Figure 6.5	New rendering.....	84
Figure 6.6	New stitching.....	84
Figure 6.7	Original rendering.....	85
Figure 6.8	Original stitching.....	85
Figure 6.9	Proposed rendering by Larsen	86
Figure 6.10	Proposed stitching by Larsen	86
Figure 6.11	New rendering.....	87
Figure 6.12	New stitching.....	87
Figure 6.13	Original rendering.....	88
Figure 6.14	Original stitching.....	88
Figure 6.15	Proposed rendering by Larsen	89
Figure 6.16	Proposed stitching by Larsen	89
Figure 6.17	New rendering.....	90
Figure 6.18	New Stitching	90

Abstract

Managing and Rendering Large Environments in Games

Hazrat Pradipta Ranjali

Master of Engineering

School of Computer Engineering

Nanyang Technological University

2010

Terrain rendering is one of the most challenging interactive visualizations in a game application. It contains a large number of triangles which may slow down the rendering process if there is no special algorithm implemented to reduce the triangles number.

There are a lot of dynamic terrain algorithms to speed up the rendering process. Most of them have only one concern: to reduce the number of triangles to be sent to GPU for processing. However, game applications nowadays are not only concerned with frame rate, they also aim to achieve good visual quality and simulate the real world.

This report presents a comparison, analysis, and characterisation of various type of terrain rendering algorithms which support terrain collision detection. In GeoMipMap algorithm, data streaming is improved by pre-computing necessary data. Visual artefact is fixed by considering triangle angles during triangulation. In GeoClipMap algorithm, GPU terrain collision is implemented to handle large number of queries.

The performance of these improved methods is then tested using heightmap data from one commercial game engine. The number of data streaming

invocation calls and the time required decrease significantly. Visual artefact does not appear in the rendering. And the performance of GPU terrain collision detection is almost unaffected by the number of queries.

Chapter 1 Introduction

This chapter gives a brief introduction to this project, the scope and objectives, and finally the structure of this report.

1.1 Motivation

Terrain rendering algorithms provide many kinds of methods to render the surface area of the real world (earth surface) or an imaginary world (fictional). It is one of the interesting topics of research in graphics area, because terrain cannot be broken down into some small parts whose complexity can be set independently.

Terrain is widely used in many 3D applications, especially outdoor graphics environment, such as 3D games, virtual reality application, and simulation (e.g. flight simulator). In such scenes, terrain may contain millions of triangles. Although current graphics cards have ability to render millions of triangles per frame, many applications continue to push the limits for graphics card, as the demand for better visual quality of the applications increases[1]. Some developers improve the graphics rendering to make it more realistic by increasing the number of polygons. As a result, the gap between the graphics card performance and the necessity to display better visual quality is likely to become wider in the future.

Figure 1.1 Terrain rendering on Battlefield games[2]

Figure 1.1 shows a terrain rendering which contains millions of triangles to display high quality image in an outdoor environment. Compared to the other objects, large terrain consists of a lot more triangles. If the brute force approach is used to render such scene, the application will have low frame rate per second (fps) and the game is not playable.

Thus, it is necessary to design an algorithm to reduce the number of triangles to be rendered and to achieve real-time frame rate. There are many available methods, varying from pre-processed algorithms to real-time ones. These methods can accomplish the purpose of rendering. But, they introduce some new problems such as expensive CPU computation, visual artefact, and complex physics implementation.

Some terrain rendering algorithms try to reduce the number of triangles as much as possible. While they have fewer number of triangles compared to the other algorithms, they require expensive CPU computation. If the application

only renders the terrain, high frame rate could be achieved. But, if these algorithms are implemented as part of a game engine, they may cause low frame rate as several elements such as physics and animation of the game engine also require a lot of CPU computation. Hence, it is desirable to design a terrain rendering algorithm with little CPU overhead.

In run time, terrain rendering algorithms compute level of details for specific parts of the terrain and do re-triangulation. During this process, visual artefacts such as “popping effect” and crack may occur. These artefacts are visible if the camera distance is near to them. Popping is a common problem that occurs with discrete level of detail algorithms when vertices within a mesh are modified in successive frames. Crack occurs when re-triangulation process removes one or more shared vertices from one instance which cause edges to be no longer common to both neighbours. To remove these artefacts, some modifications such as morphing and stitching are required. However, imperfect solution may cause visual artefact to appear under certain condition.

When people play a game, they expect it to reflect the real world, so that they can immerse into the game’s world. In an outdoor environment, there should be a system to check if object collides with the terrain, to prevent them from sinking below the surface or flying. One technique to achieve this is to have a terrain collision detection system, which supports the intersection test even when the character is moving, in the game engine.

In computer games, collision detection is one common module that can be found in most of games engine since the majority of games involve simulating

movements. There are three main concerns to decide *if*, *when*, and *where* two or more objects collide with each other[3]. “If” includes the test whether the objects intersect. “When” determines the time of collision. And “where” decides the location an object collides with another object.

An application such as animation does not require real-time computation for its collision system. But, a game system has a great demand for efficiency in collision detection system to achieve real time performance. Game has very tight time constraints because the frame rate must be maintained at 30-60 frames per second (fps). Collision detection system is only a small part of the code in a game engine. Nevertheless, it is among 10 percent of the code which takes about 90 percent of the CPU processing time in some game engine. Hence, it is fundamental to optimize the system to avoid it becoming a bottleneck in the rendering loop.

There are some considerations in the implementation of terrain collision detection: large number of triangles, irregular structures, and triangulations that keep changing for each frame. Besides that, the optimization for object-object intersection test such as sweep and prune[4] cannot be applied for object-terrain collision detection.

In this project, fast terrain collision detection is implemented for various types of terrain rendering algorithms. Some optimizations are made to reduce the computation time. Finally, GPU terrain collision detection is implemented and compared to the performance with CPU implementation.

1.2 Scope and Objectives

Terrain is a three dimensional representation of land surface, which carries the elevation, slope, and orientation of the terrain. In computer application, it is used in many outdoor games such as car racing, flight simulator, and virtual reality which enable the user to interact with a computer-simulated environment.

There are two terrain data structures: regular grid (heightmap) and Triangulated Irregular Network (TIN). To render large terrain, heightmap is preferable because it is easy to implement data streaming from hard disk and it allows triangles modification (for Level of Detail/LOD) in the run time. For heightmap itself, the terrain rendering algorithms can be further classified into two categories: CPU oriented and GPU oriented.

CPU oriented algorithms make the assumption that the GPU has a very limited capability to render the large number of triangle. This assumption applies to the old graphics card at the time the algorithms were invented (before 1998). Thus, it tries to reduce the number of triangles as much as possible. [5-6] belong to this category.

On the other hand, GPU oriented algorithms make full use of modern GPU rendering that can process a large amount of triangles per frame. Thus, rather than trying to achieve a minimum number of triangles, it shifts some works from CPU to GPU to reduce the amount of CPU overhead. [7-8] belong to this

category. Going further, [9] and [10] implemented the GeoClipMap algorithm on GPU which offloads work from the CPU.

The **main contributions** of this project are: various types of terrain rendering algorithms are investigated for comparison to determine the best algorithm. Experiments are conducted to gather information on frame rate, number of triangles, CPU timing, terrain collision timing, and visual quality.

The visual artefact that appears during triangulation is fixed by limiting LOD differences on the neighbouring patches. The current stitching implementation is still prone to error due to thin triangles. It is improved by considering triangle angles.

Data streaming is implemented to render large terrain. The concern is not only about the data retrieval from hard disk, but also level of detail pre-computation which is required on some algorithms. Besides, the area which is not covered by the terrain data needs to be masked out.

Terrain collision detection is implemented to handle large number of queries for various types of terrain rendering algorithms. Given a terrain which consists of elevation data, the system computes if an object collides with the terrain. This type of Boolean query is commonly used for collision detection. In games, precision can be used as trade off to gain speed. So, approximation is often needed to achieve tolerable frame rates. The answer to query is given with some error tolerance.

For an object that collides with the terrain, the system computes the penetration depth. It is the shortest distance that separates the object and the terrain. If the distance of an object to the terrain is less than or equal to zero, then the object is intersecting. But, if the value is greater than zero, then it is not. When an object is not intersecting with the terrain, the distance value between the object and terrain could be used for collision prediction. For moving objects, computing the time for collision prediction is often known as Estimated Time of Arrival or Time of Impact[3].

Some terrain rendering algorithms nowadays are able to perform the rendering process entirely in the GPU. In these kinds of algorithms, the elevation data is stored on the GPU. As a consequence, terrain collision detection must be entirely performed on the GPU, or the data must be duplicated on the CPU memory. In this project, GPU terrain collision system for GeoClipMap algorithms is implemented using DirectX 9 shader. An experiment is made to compare the computation time between CPU and GPU implementations.

1.3 Overview

The main topic of this report is about terrain rendering and collision detection. The background and foundation for this involves terrain data structure, various LOD rendering algorithms, GPU Programming, and Scene Graph. These subjects are discussed in Chapter 2: “Background”. Chapter 3: “Terrain Implementation” explains the implementation of large terrain rendering. Some modifications of the current implementation are explained here, which

covers data streaming, improved stitching method, and GPU terrain collision. Experiments are conducted to compare the performance of various algorithms which are discussed in Chapter 4: “Experiment and Analysis”. Besides that, it also explains the performance of the proposed implementation. Chapter 5: “Summary and Conclusion” describes the summary and result of this project. Appendix is added at the end of the chapter to provide some images of visual artefact in GeoMipMap.

Chapter 2 Background

This chapter describes some relevant topics which can help in understanding the content of the overall report. First, it explains how the terrain model is rendered using different data structure and various LOD rendering algorithms. Then, it introduces the general concept of GPUs. And lastly, it explains how scene graph reduces computation.

2.1 Terrain Data Structure

Terrain in the real world is a continuous surface, and even a small part of it may contain infinite number of elevation data value. It is not possible to store all these data because the computer system has limited resource, especially storage space. So, it is enough to fetch “satisfactory” representation of the terrain for processing and storage efficiency.

Digitization is a transformation of analog information (in this case the terrain data) into digital form. It is a mapping from continuous 2D sets of real objects to discrete sets[11] and allows all kinds of information in all formats to be processed with the same efficiency and mixed together[12]. The digital representation of terrain consists of a set of discrete points or samples which approximate the original terrain elevation.

Figure 2.1 TIN (left) and Regular grid (right) data structure[13]

Figure 2.1 shows two kinds of terrain digitization methods based on the sampling interval. The left figure shows TIN, which uses the knowledge of the elevation surface to select points that contain maximum amount of information, such as valley floors, ridges, and pits. And the right figure shows the regular grid or heightmap. It uses regular sampling points with equal increments in the x and y coordinates.

	TIN	Heightmap
Spatial organization	Non-uniform	Uniform
Polygon number	Less	More
Polygon density	More polygons for rough area	Equal density for the whole surface
Indexing	No	Yes
Number of possible elevations	Unlimited	Depend on the number of bit

Table 2.1 Comparison between TIN and heightmap[14]

Table 2.1 shows the comparison between TIN and heightmap. For a given level of approximation, TIN produce less polygons compared to heightmap, but in terms of data size, it is less compact since each point needs to store three coordinates. Moreover, neighbours information cannot easily be determined from the points alone. On the other hand, heightmap representation uses only one-third of the storage required for the same number of data as it only stores elevation data. Besides, neighbours information is stored implicitly from indexing information. As a result, TIN generally requires much more storage allocation compared to heightmap[15].

TIN uses varying space between vertices in the triangulations. Hence, some part of terrain contains more triangles than the others. In general, rough surface contains more information than flat surface. Delaunay triangulation [16-17] are used as a base algorithm to construct TIN. [18-19] constructs TIN data with certain “surface specific” points and critical lines to represent important terrain feature such as ridges, roads, and channels. TIN emphasizes terrain generation. The main concern of TIN is to reduce the number of triangles as much as possible for certain error value. It is used to represent a terrain surface before the data is loaded in the game.

Figure 2.2 shows the input and output of TIN algorithm. There are three common inputs for this algorithm: Heightmap, Control Map, and User Constraints. Heightmap is used as a base for terrain elevation data. Control Map is used to indicate the areas in the terrain which require more details of information. And User Constraints is a set of parameters such as terrain size,

maximum height, minimum area, maximum error value, minimum angle, and accuracy of the algorithm.

Figure 2.2 Input and output of TIN [18]

In general, TIN is generated from heightmap. Heightmap is a gray-scale image used to store an array of different surface elevation data to be displayed in 3D rendering. The data is sampled from the intersections of two orthogonal lines which is regularly-spaced. The altitude of the terrain is measured in this

intersection and stored as a 2D array. However, a value in the array does not indicate the absolute height for a given point because the data is stored as 8-bit or 16-bit format. Instead, it represents the height relative to the rest of the map by scaling it based on the minimum and maximum values of the terrain.

There is a new variant of heightmap, called isometric heightfield[20]. It is built with 60 degree axes to produce equilateral tessellation. It improves the quality of rendering by using symmetric tessellation data structure. However, there is not much level of detail algorithms for this variant up to date.

Figure 2.3 Heightmap and output

Figure 2.3 shows an example of heightmap and its rendering output. In this example, the number of triangles is relatively small. When it contains large number of triangles, it may slowdown the rendering process if there is no special algorithm implemented to reduce the triangles number. So, level of detail (LOD) is used for rendering. Basically, the algorithms increase the number of triangles for polygons near to the camera and reduce the number when they are far away.

2.2 LOD Rendering Algorithms

TIN optimizes the mesh before it is loaded on the game and currently there is no real-time computation for this algorithm. On the other hand, terrain rendering with heightmap allows real-time computation to reduce the number of triangles when the application is running. The main focus of the terrain rendering with heightmap algorithm is to provide certain level of detail on the terrain geometry with minimum number of triangles given an error tolerance value. Some metrics are used to determine where the detail is required.

For a large terrain, heightmap is more suitable, since Level of Detail simplification will result in far less triangles number. And scene graph implementation further reduces the computation. But, LOD implementation introduces some new problem: “Popping effect”, visible crack, and physics implementation.

“Popping effect” occurs when there is a transition from one LOD level to another one. Crack might be visible along the edge between two adjacent areas (block) which have different LOD levels. And physics implementation (collision detection) may be computationally expensive for some algorithms.

Aside from these problems, there are some advantages of terrain rendering using heightmap, such as:

1. It is possible to have terrain deformation. In some scenario, there may be an event which modifies the terrain height, such as an explosion or

the presence of a vehicle track[21-23]. Hence, the algorithm must re-compute the triangulation fast to handle terrain deformation. Since TIN is computationally expensive, it is impossible to have such feature.

2. Real-time procedural terrain generation[24]. Procedural terrain means that the terrain data is created on the fly in real-time. The data itself uses some algorithms and mathematical functions, for example: Perlin noise[25] and Voronoi diagram[26-27]. One main advantage of using this technique: it is extremely compact. Unlimited number of terrains can be created without requiring large storage requirements. And it can be parameterized to create a specific terrain, for example bumpy or flat terrain. Since the terrain data is created in real-time, terrain rendering must be in real-time as well. One example of game application which uses this technique is Tribal Trouble.
3. It is possible to set different visual quality for different computer specification in real-time. If the application is used for Personal Computer (PC), it may need to be configured to meet specific hardware specification. Hence, high-end PC can display very detailed and sophisticated terrain, while PC with lower specification has lower quality display. Varying visual qualities can be achieved by setting different error tolerance values.

There are several kinds of terrain LOD renderings, such as Continuous LOD, ROAM, GeoMipMaps, and GeoClipMaps. For more details on these methods, please refer to [28-29]. Out of these algorithms, there is a set of defined rules to achieve high quality rendering, such as:

1. Minimum number of triangles to be rendered.

In a game application, terrain engine has a time limit to render the terrain. Hence, a good algorithm is one which reduces a great number of triangles with no or little loss in image quality. However, some algorithms may have “inconsistent” performance in rendering; depending on the terrain topology, viewer (camera) angle and viewer position. They may need different period of time for each frame to render the terrain. It can be caused by frustum culling of the scene graph. But, it is mostly affected by the algorithm itself. A good algorithm is one which has a parameter that can be set to determine the maximum time required for rendering or the maximum number of triangles for each frame.

2. Meshes quality.

Some algorithms may use Level of Detail, either discrete or continuous in their implementation. If it is discrete LOD, then there is a high chance of undesirable “Popping effect” appearing in the visual display, which reduces the visual quality. It is expected that a mesh changes smoothly in between successive frames especially when it shifts from one LOD level to another one and maintains the frame rate at the same time. Hence, to reduce popping effect, vertex morphing must be implemented.

3. Real-time execution.

A dynamic terrain processes a large number of triangles. So, the algorithms must reduce the triangles fast enough in order to meet time limit requirement. And in general, the algorithm selected must be easy to understand and implement.

4. Fast query.

The terrain is not used only for display purpose. It is also used for some query information, for example collision detection. And the query result must be consistent with the rendering. It means that the computation must be based on the triangulation used in rendering, so the object does not appear flying or sinking into the terrain.

2.2.1 Continuous LOD for Height Fields

Lindstrom et al. [5] introduced the first real-time LOD terrain rendering algorithm in 1996. The method is used to render high fidelity terrain based on a screen-space error metric to limit the maximum error of the projected terrain. The screen-space error metric introduced by Lindstrom is widely used in the next generation algorithms introduced by other authors, such as ROAM, GeoMipMap, GeoClipMap, etc. The metric is an important measure to determine whether vertices should be removed during rendering process.

In this algorithm, the terrain is divided into blocks and processed as Quad Tree using a bottom-up refinement strategy. There are two steps to render the

meshes. First, a coarse-grained simplification selects discrete LOD for each block of the surface meshes. Second, a fine-grained re-triangulation further simplifies the geometry which considers each vertex within selected blocks for removal.

On a fine-grained re-triangulation, some vertices are removed, so that some small triangles can be substituted by fewer bigger triangles. When a vertex is removed, an edge is created between two of its neighbours. There is a condition for two triangles to be fused into one bigger triangle. The condition is described as the amount of slope change between two triangles. The slope is measured as the distance between the discarded vertex and the new edge created during the process. The slope change value must be less than certain threshold, τ , so that the triangles can be fused.

Figure 2.4 Geometric representation of delta values[5]

Take a look at Figure 2.4. Consider two triangles: $\triangle GED$ and $\triangle DEA$. These two triangles can only be fused into one triangle $\triangle GEA$, if δ_D value is less than τ . As

the value of δ_D increases (higher slope change), the chance of merging decreases.

Now consider that $\triangle ABE$ and $\triangle BEC$ are also merged into $\triangle AEC$. In this case, $\triangle GEA$ can be further merged together with $\triangle AEC$ if δ_E value is less than τ . So, this merging process can only be done if both pairs ($\triangle GEA$, $\triangle AEC$) have been fused. This example shows that there is a dependency during the merging process since the fusion can only happen if triangles in the triangle pair exist on the same level. This merging process is repeated until no further simplification is possible.

However, this process cannot be done for each individual triangle since it takes too much computation. So, in the first step, coarse-grained simplification is executed using a terrain block.

A block is defined as a set of adjacent elevation points within a rectangular area. The dimension of each block in this algorithm must be $2^n + 1 \times 2^n + 1$, for $n > 0$. Quad Tree is used as a data structure which holds the pointer for all the blocks to cover the entire terrain. The process starts by calculating maximum delta value δ (slope change) for each lowest level vertex within a block. The calculation is repeated for n Levels of Detail. These maximum delta values are then stored in memory and used to determine the discrete LOD. After selecting the discrete LOD, those 'unnecessary' vertices can be removed quickly without extra computation, because the maximum value of delta projection is known before hand.

In order to create lower resolution blocks, four ‘sibling’ blocks can be grouped together by discarding every other vertex on alternate row and column, as illustrated in the picture below

Figure 2.5 Obtaining lower resolution blocks

Figure 2.5 shows the process of obtaining lower resolution blocks. On the left, there are four blocks; each has a size 3 x 3. And on the right, the four blocks are merged into one bigger block with the same size.

In conclusion, there are some advantages and disadvantages of using this algorithm. The main advantage of using this algorithm is high frame rates because coarse-grained simplification reduces a large number of triangles without imposing high computation cost. However, the proposed algorithm suffers from visual artefact. By using block to split the terrain, T-junction and crack may appear at the surface in between neighbouring blocks, as shown in Figure 2.6. This is one main problem when rendering LOD terrain, especially if it uses tile-based approach[30]. So, special care must be taken to solve this problem which may increase overall algorithmic complexity. Besides that, the algorithm does not support triangle-count parameter. There is no way to

obtain the maximum number of triangle during rendering process. As result, the performance (frame rate) is different for each frame.

Figure 2.6 T-Junction and crack

2.2.2 ROAM

In 1997, Duhaineau et al.[6] proposed ROAM (Real-time Optimally Adapting Meshes) algorithm which renders dynamic terrain based on Triangle Bin-Tree. Bin-Tree is used in conjunction with split-merge operations to obtain optimal polygonization using top-down approach. One interesting feature of this implementation is the number of processed triangles per frame is far less than the total mesh rendered. It happened because the algorithm incrementally refined the triangles using a priority queue.

The main operation of this algorithm is splitting and merging the triangle. In order to support these two operations, right-isoscele triangle is used. Take a look at Figure 2.7. The root of Triangle Bin-Tree ($l=0$) lays the coarsest level of this triangle. And on the next level ($l=1$), the children are defined by

performing bisection on the longest edge (diagonal). This process is repeated to construct the rest of the Bin-Tree.

Figure 2.7 Triangle Bin-Tree level

There are two kinds of split-merge operation as illustrated on Figure 2.8. First, diamond triangulation where two adjacent triangles have the same level (Figure 2.8 left). The second one is the triangulation of different level adjacent triangles (Figure 2.8 right).

Figure 2.8 Split-Merge operation

If two adjacent triangles have the same level, then the split and merge operation is quite simple. Split creates a new vertex, while merge deletes the vertex. But, if the level is different (as illustrated in Figure 2.8 right), the process cannot be executed immediately. To split triangle T, triangle Tb must

be split first which further requires chain reaction of another split. This forced split is compulsory to avoid visual artefact such as T-junction, and to create a seamless surface.

There are two conditions in split and merge operations, that is the terrain roughness and desired frame rate (or time limit). The terrain roughness is used in the vertex removal. It computes slope changes from height deviation before and after vertex removal. However, such computation is quite expensive if there are a lot of vertices to be processed. So, some operations can be delayed if predetermined frame rate is not met; at the cost of lower visual fidelity. This trade-off is quite useful to meet some specific criteria, such as minimum frame rate for game application.

In conclusion, ROAM has the advantages of frame-to-frame coherence, optimal triangulation, and support of triangles count. There are several variations and improvement of ROAM, such as [31-33] to reduce the computation by using diamond data structure and AGP (Accelerated Graphics Port) chunking. However, the nature of this algorithm takes a lot of CPU computation, especially since it performs per-triangle basis to calculate the priorities of all triangles for every frame and to execute split-merge operation. Because of this, the algorithm is not suitable for modern graphics processor.

2.2.3 Geometrical MipMapping

De Boer[8] published an article on the internet describing a novel approach of rendering LOD terrain, called Geometrical MipMapping (GeoMipMaps)

algorithm which is similar to Lindstrom algorithm[5] in certain aspects, such as terrain blocks and screen-space error metrics. The main difference is that it reduces CPU overhead by pushing more triangles to rendering pipeline since the terrain rendering are not processed per-triangle, but per-block.

Figure 2.9 Texture mipmapping

This algorithm is inspired by texture mipmapping technique[34]. A texture mipmap is an optimized set of raster images which is constructed from a main texture. As illustrated in Figure 2.9, each image is recursively built from the main image at lower resolution, usually a power of two smaller than the previous resolution. High-detailed image is used when the viewer (camera) is near to the object, while lower-resolution ones are used as the viewer moves farther away. The graphics renderer will determine a suitable level of mipmap image and sometimes interpolate the two-nearest level images based on viewer distance. This technique is used to improve the quality of texture rendering and increase the rendering speed at the cost of more memory requirements.

Using texture mipmapping technique alone is not enough for LOD terrain rendering, especially since the terrain size is very large. So, the terrain must be divided into a set of smaller quadratic rectangular shape called block; each has an equal size of the form $2^n + 1$, for $n > 0$. Similar to texture mipmapping, each terrain block has a set of different level of details. For size $2^n + 1$, a block can be down sampled up to $(n-1)$ levels and all of them cover the same area. A block of level $(l+1)$ uses half of the number of vertices of a block of level l .

There must be some condition in choosing the appropriate level for each block to maintain the quality of rendering. Shifting from one level to another one may cause popping artefacts since there are vertex additions or removals on terrain blocks which in turn change the slope of terrain. This popping is not noticeable when the distance of viewer and the geometry changes exceeds threshold value D_n , because of the perspective view. D_n is minimum distance at which level of details n can be used to reduce the triangles number. This value is calculated based on maximum geometric height-change in a block during vertex removal. Now, take a look again on Figure 2.4. Assume that vertex B is to be removed. As result, there is a slope change (geometric height-change) whose value is δ_B . Since each level contains several vertex removals, there exists several value of slope change. To simplify the calculation, take the maximum value of this slope change as a worst-case scenario. The assumption is: if this maximum value is lower than some threshold τ (which represents pixel error) then all of the slope changes in this particular level are smaller than τ and this level of detail can be used without affecting much rendering quality.

Calculating geometric height-change for every vertex removal in every frame is computationally expensive. So, instead of that, the algorithm pre-computes the value of Dn for each level of detail in each block and stores the result in a memory. It decides the appropriate level in a block by comparing the viewer distance to the centre of block with the value of Dn . The pseudo-code to choose level of details is described below.

```

For i = 0 to (n - 1)
    If  $Dn > \text{viewer distance}$ 
        return i
End For
Return n-1

```

To pre-calculate Dn , take a look on the equation below which describes the relation between Dn and τ .

$$Dn = |\delta| \cdot C$$

$$C = \frac{NC \times Vres}{2 \times |t| \times \tau}$$

δ is maximum geometric height-change for the current level, C is a constant value, NC = near clipping plane, $Vres$ = vertical screen resolution in pixels, and t = top coordinate of the near clipping plane.

Another advantage of using terrain block is for view-frustum culling. Culling is used to remove blocks invisible from the current viewer angle point since they need not be rendered. To make it even faster, the algorithm use Quad-Tree to

process the culling operation. The Quad-Tree is constructed at loading time, before rendering.

Figure 2.10 Adjacent blocks of different level of details

However, LOD terrain block introduces two new problems: T-junction and ‘popping effect’. T-junctions can occur at the shared borders if the adjacent blocks have different levels of details, as shown in Figure 2.10. A crack may appear out of this T-junctions because higher level block uses fewer vertices at the edge.

There are two ways to solve this problem: skirts and stitching[35]. Skirts are used by surrounding each block with a strip of triangles. The size is calculated based on the maximum geometric change. Since it can be pre-computed and the algorithm is simple, there is no much penalty on runtime rendering. It can mask any cracks with reasonable geometric error by using vertical triangle around the edge of block. Skirts are simply ‘hiding’ the cracks and it does not really remove it.

Figure 2.11 Solving T-junction by stitching

Stitching is used to create seamless polygon surrounding the block by removing the cracks. There are two approaches in stitching: modifying the triangulation or adding extra mesh. GeoMipMap implementation uses the first approach by omitting the ‘unused vertices’ and modifying the vertices connectivity on the edge. This process is illustrated in Figure 2.11.

‘Popping effect’ happens during the switching of level of details. It can be solved by implementing Geomorphing using Trilinear Filtering. It interpolates the elevation data value of the vertices to be removed at the next level. However, this process adds workload to the CPU. To reduce the CPU workload, Wagner[36] proposed the implementation of Geomorphing using vertex shader. Other extensions of this algorithm are described in [37-39]. These extensions modify the computation of error metric, triangulation and the implementation for large textures.

2.2.4 Geometry Clipmaps

Losasso and Hoppe[7] proposed Geometry Clipmaps (GeoClipMap) algorithm which uses concentric regions as level of details. The approach is similar to Texture Clipmap by Tanner et al[40] which uses rings of texture resolution, as shown in Figure 2.12.

Figure 2.12 Rings of texture resolution [40]

Geometry clipmaps are different from the previous approaches, as its level of details in the terrain does not depend on the terrain topology (geometry). The author argued that the computation for LOD using terrain topology increases the complexity of the algorithm and consumes extra memory. Instead, the algorithm only needs to depend on the viewer point distance, resulting in a simple data structure.

The key point of this algorithm is that it uses a viewer-centric nested regular grid when processing the geometry. Several region levels (clipmap) are defined; each has the same number of $n \times n$ vertices, but having different areas. The value of n determines the screen-space triangle size. This relation can be described in the following equation:

$$n = (1.25) \times \frac{W}{s \times \tan \frac{\varphi}{2}}$$

where s = screen-space triangle size, W = window size, n = clipmap size, and φ = field of view.

Figure 2.13 Geoclipmap regions (left) and terrain (right)

Figure 2.13 (left) shows three levels of geoclipmap regions. The successive level has lower resolution, a power of two smaller than the previous one. To avoid overlapping vertices during terrain rendering, the 'hollow regions' are created by excluding the next level regions from the current one, as illustrated on Figure 2.13 (right). The clipmap level is built based on the distance to the viewer position. The geometry region near to the viewer has higher resolution. As the viewer distance increases, the resolution is reduced. By using this method, the rendering process can be executed fast.

When the viewer moves, the regions are updated by shifting them such that they are centred at the viewer. To allow efficient (incremental) update, only a small portion of array data is replaced using Toroidal addressing. Toroidal addressing is a method to store 2D array of data using modulo operation by adding new-visible data (area) to replace old-invisible data (area). Figure 2.14 explains this updating process. When the viewer moves to the north-east direction, there are some data which do not need to be updated, since the area is the same as before. Only a small portion of old data needs to be replaced with the new one. The new data on the top (A, B, and C) of the image replace the data at the bottom. And the data on the right (E, F, and G) replace the data at the left.

Figure 2.14 Toroidal addressing

Having a number of different region levels, this algorithm also suffers from popping effect and T-junction due to the gap at the boundaries. To solve these, vertices near the outer boundaries are morphed as a transition to the coarser level. The morph value α depends on the transition width w and camera position v as described in this equation:

$$\alpha_x = \min \left(\max \left(\left(|x - v_x| - \left(\frac{x_{\max} - x_{\min}}{2} - w - 1 \right) \right) / w, 0 \right), 1 \right)$$

$$\alpha_y = \min \left(\max \left(\left(|y - v_y| - \left(\frac{y_{\max} - y_{\min}}{2} - w - 1 \right) \right) / w, 0 \right), 1 \right)$$

$$\alpha = \max(\alpha_x, \alpha_y)$$

By using this value, and given that z = height of current level, z_c = height of next coarser level, the new morphed height value is defined as:

$$z' = (1 - \alpha)z + \alpha z_c$$

Although morphing can fill in the gap at the boundaries, T-junction may still exist during rendering. To completely remove it, zero area triangles are rendered to stitch the boundary between adjacent levels, as shown in Figure 2.15.

Figure 2.15 Zero area triangles (red)

2.3 GPU Programming

Before early 1990s, interactive 3D graphics was not commonly used by people since there is not much improvement in graphics hardware. However, just within one decade, almost everyone has experienced some high-performance, visually rich, and interactive 3D display because of dedicated GPU on nearly every computers and game consoles. For example: the mass production of video game consoles such as PlayStation and Nintendo 64 which have integrated 3D graphic hardware. This trend is inevitable due to the demand for more advanced video games and graphics technology.

Figure 2.16 Direct3D Graphics Pipeline[41]

Figure 2.16 illustrates the basic processing pipeline in Direct3D Graphics. The input consists of vertex data, primitive data, and texture surface. Vertex data is the model vertices in world coordinate system. Primitive data includes points, lines, triangles, and polygons which are represented as index buffer. And texture surface is the texture coordinate given by the IDirect3DTexture9 interface. The final output of this pipeline is the pixel rendering which displays certain colour information.

The structure of modern GPUs is highly parallel to support the following operations:

1. Texture mapping
2. Polygon rendering
3. Geometric transformation, such as rotation, translation, and scaling
4. Matrix, vector, floating point operation
5. Programmable shader

Inside a GPU, there is a specialized processing chip for rendering 2D images or 3D models. If the application needs to display more realistic image or video, it must be rendered on more advanced and faster GPUs. For this reason, nowadays latest computer games, which have high detail and realistic visual display, require the users to upgrade their PCs to the latest hardware.

In the recent years, CPU manufacturers barely improve CPU speed according to Moore's law. When the speed is capped to about 3GHz, they add more cores inside the processor. However, it is a different case with GPU. In the last decade, the computing technologies in GPU increase tremendously, exceeding Moore's law[42]. In most cases, modern GPUs have theoretical peak power higher than CPUs. Adding the introduction of PCI Express which improves data transfer, there is a tendency of shifting some parallel computation from CPUs to GPUs.

Nowadays, there are many computational research topics on the use of GPUs for non-graphical calculations, especially since GPUs is faster than CPUs for

the floating point, matrix, and vector operations. General-Purpose computing on Graphics Processing Units (GPGPU) is the technique to improve the performance of computation in applications by using GPU programming instead of CPU. The important step of GPGPU is the introduction of programmable graphics hardware and assembly language to specify a set of instruction to run on each vertex[43]. Besides that, it is also supported by graphics card manufacturer. For example, Nvidia introduced CUDA as a computing engine that is accessible through standard programming language. It enables developers to code a program for execution on the GPU.

In games, some computations can be done in parallel to achieve better performance. However, CPUs generally support serial programming models only and do not expose data parallelism since they process one data at a time[44]. The addition of special instruction set such as Intel's SSE and multithreading support can achieve parallelism, but it is still limited compared to GPU hardware.

GPUs are different from general-purpose CPUs. They are a special hardware designed to achieve efficiency for specific functions, especially in rendering. Their support for parallelism can be used for general computation to maximize the throughput. There are some ways to implement GPU programming, such as C for Graphics (Cg), Compute Unified Device Architecture (CUDA), OpenGL Shading Language (GLSL), and DirectX High Level Shader (or Shading) Language (HLSL).

One main problem in GPU programming for general computation is its difficulty to use. In the first place, GPUs are designed to accommodate graphic rendering. Hence, the programming model is tightly constrained and it is tied to the graphic pipeline. As such, programmers can't simply port code from CPUs to GPUs.

In GPU programming, a set of data of the same data type is represented as a stream and processed in parallel. In this sense, GPUs are processors running a kernel that operates on entire streams at once. The computation of one element is independent from another element. There is no static data and read-write buffer. Hence, all of the data required must be known before hand.

The input data can be obtained from several ways. It can be in the form of vertex buffer, texture, or global variable. For general computation, programmers usually put the main data from CPU into texture as it naturally map into grids. Some non-grid input such as 1D array can be mapped into grids. Texture lookups are then used to access data structures and streams. During execution, kernels process all data provided by texture for general computation. And the result is fed into a render target. Since the render target can be treated as a texture, the output can be retrieved back to CPU.

Chapter 2**Background**

```
// CPU program
for(int i=0; i<N; i++)
{
    output[i].x = dataX[i].x * alpha + dataY[i].x;
    output[i].y = dataX[i].y * alpha + dataY[i].y;
    output[i].z = dataX[i].z * alpha + dataY[i].z;
    output[i].w = dataX[i].w * alpha + dataY[i].w;
}
```

```
// GPU program
float3 fCompute(
    float2 inTexX : TEXCOORD0,
    float2 inTexY : TEXCOORD1
)
{
    float4 dataX = tex2D(SAMPLERX, inTexX).xyzw;
    float4 dataY = tex2D(SAMPLERY, inTexY).xyzw;
    return dataX * alpha + dataY;
}
```

The code above shows the difference between CPU and GPU programming. In CPU programming, “for loop” is used to process data with the same instruction. But, since GPU does parallel processing, “for loop” instruction is removed. Instead, it fetches input data in texture using “tex2D” command. The other difference is unit operation. GPU operates in a vector unit: a single operation can be used for four values at once. Since *dataX* and *dataY* are vectors whose components used the same functions, then it can be combined into a single instruction.

2.4 Scene Graph

Scene graph (or Scene Management) is a data structure to arrange spatial representation of objects in a scene. It is used to reduce the processing computation of objects, for instance: object collision and object rendering. For a large number of objects, rendering takes a lot of computation time. So, the best way to draw a scene is to avoid drawing “unnecessary” objects using object culling. For example, Lindstrom[5] uses Quad-Tree data structure for block partitioning of the height field dataset. And Miller[45] use Quad-Tree in the pre-processed height field. It breaks down a terrain into a pyramid of resolutions. And for each level in the hierarchy, the terrain is divided into 2D array of cells.

For object culling, an application usually uses either Bounding Box or Bounding Sphere, with the assumption that the culling test is a lot less expensive to compute than rendering computation. A sphere is easy to implement and inexpensive, but it is generally less accurate in giving test results. On the other hand, a box takes more computation, but it is more accurate. Hence, the overall minimum cost computation to achieve the best result very much depends on the culling test and rendering cost.

There are three kinds of scene graph algorithms:

1. Spatial Coherency: groups the objects based on their location. It is the most natural choice algorithm, since it is intuitive. Quadtree and

Octree[46] belong to this category, which is free to use. While Grid-based loose octree (GLOtree)[47] algorithm is patented in US.

2. Render-state Coherency: groups the objects based on their render state. In graphics rendering programming, render state change should be minimized to provide faster frame rate per second (fps). [48] explains more about this technique.
3. The combination of the two techniques above. First, sorts the objects based on spatial hierarchy, then followed by render state.

For terrain collision, scene graph is used to reduce the amount of geometry/polygon to be processed for further computation. Although a terrain is a 3D object, the scene graph used usually is a 2D scene graph, since the occlusion test may ignore the top and bottom side (terrain height) in the bounding box of terrain parts. It uses a divide-and-conquer method by breaking down the terrain into many parts.

Chapter 3 Terrain Implementation

This chapter explains the proposed implementation of large terrain rendering. First, it describes the problems of current algorithms. Then, some implementations are proposed to obtain the desired traits of large terrain rendering algorithms. The modifications are explained which include data streaming, stitching, and GPU terrain collision.

3.1 General Problem

One of the interests of terrain visualization comes from flight simulator research to improve the visual aspect of training environment for pilots. It displays low-detailed terrain in far away distance due to aerial view and requires high-detailed composition when an airplane is approaching the ground. In order to provide high quality detail and large coverage, level of detail algorithms were developed. The necessity to meet these requirements has attracted researchers in exploring many multi-resolution techniques for terrain rendering.

In this scenario, rendering is not only concerned with the number of polygons. Similar to the other objects in a game application, a terrain is expected to provide fast computation and rendering, good visual quality, and interactivity to improve game experience. Yet, most of current algorithms only explain polygon simplification for the terrain.

Fast computation and rendering can be obtained if algorithms use minimum CPU computation and produce minimum number of triangles. To achieve these, the algorithms must provide frame-to-frame coherence, a method to reuse vertices visibility computation from previous frames efficiently during continuous visualizations[49]. Besides that, there is a necessity to process vertices in batch to reduce the number of computation and drawing calls. This is important as graphics card nowadays is optimized for large batches of primitives. It is better to process more polygons in a single call. Although the current graphics card can process up to millions of triangles per frame, the algorithms are still expected to produce minimum number of triangles to reduce data transfer and vertices computation.

Low memory requirement on both CPU and GPU is also one of the desired traits. In some scenario, complete data cannot be loaded on the main memory due to a large terrain. Hence, the support of data streaming is necessary as well.

Gamers expect good visual quality in a game. Some factors must be considered to achieve this: seamless triangulation and no visual artefact. Discontinuity between neighbouring triangles may result in visible crack which is not desirable. Skirts and stitching can be used to fix the crack. However, imperfect solution for this problem still causes the visual artefact to occur in certain conditions, for example, when thin triangles are used.

In the real world, most objects are affected by gravity which causes them to stay on the terrain surface. This behaviour must also be reflected in a game

application using collision detection. There are numerous ways for object collision detection, ranging in capability, focus, and complexity.

As a game has frame rate constraint, the emphasis of terrain collision implementation is placed on the computation time to achieve real-time performance even though it may cost loss in precision. Rough approximation such as bounding volume, can be used for this purpose. One main advantage is that they are fast and easy to implement[50]. In general, there are four approaches for collision detection algorithms: space-time volume intersection, swept volume interference, multiple interference detection, and trajectory parameterization[51]. However, not all of these approaches are suitable for terrain collision detection.

In general, there are some considerations for terrain collision system:

1. Number of polygons. Terrain contains more triangles compared to other objects. Even though there are numerous methods to reduce it, the number is still quite large. For outdoor environment, the number can be hundreds of thousands or millions. Graphics cards are able to process this number since they are stream processors. But, if all of these data are passed to the CPU for computation, it will become a burden. There are two phases for terrain collision system: broad phase and narrow phase. Broad phase is used to eliminate most polygons in the terrain which do not contain point P. Some methods such as bounding volume and spatial partitioning are used. Then the result which contains a

smaller set of polygon is passed to the narrow phase to compute the exact collision value.

2. Terrain rendering algorithm. Since terrain contains a lot of data, there is a need of a data structure for terrain collision. However, this data structure cannot be separated entirely from rendering algorithms. For example, if the terrain is rendered using ROAM algorithm and the collision uses the initial heightmap value, then there is high possibility that the object will appear flying or sinking. It happens when level of detail is applied at that particular point. In other words, terrain rendering determines the collision detection algorithm.

3.2 BinTrees Hierarchy

Figure 3.1 BinTree Hierarchy in ROAM

The main idea is to build a triangulation hierarchy by refinement or simplification. The algorithm allows the creation of continuous resolution

surface without the need to handle crack. Besides, the number of triangles produced is much lesser compared to other terrain Level of Detail algorithms. In this project, ROAM is implemented as a test platform as shown in Figure 3.1.

This algorithm uses a hierarchical data structure which divides the terrain up to the smallest triangle possible to produce minimum number of triangles. However, it suffers from several drawbacks, such as:

- Memory usage expands exponentially with depth. Tree data structure used in this algorithm requires at least five pointers for each triangle to maintain the neighbour relationship.
- Dynamic update computation is expensive. For large terrain, dynamic update of heightmap needs a complete re-computation of the variance over the modified area. As the area gets larger, the computation takes longer time.
- Data streaming is not possible. As the new data is added to the terrain and some part of the data is released, the overall hierarchy structure must be re-built, which is very time consuming.
- Terrain collision detection is expensive. For a collision point P, the test starts from the root and traverses down to the leaf to find a triangle which contains this point. As shown in Figure 3.2, the bounding volume of level 1 and below is a triangle which encloses some smaller triangles of the lower level. Hence, there is no overlap of bounding volume. However, the hierarchy level might be quite deep, since it depends on

the terrain size. As the number of level increases, the computation cost increases as well.

Figure 3.2 Hierarchy of collision data for ROAM

In order to solve these problems, there are two necessary modifications. First, the depth of hierarchy must be limited by modifying the rendering algorithm. The modification algorithm is based on the grid concept introduced by [52] and [53]. Second, triangle primitives must be processed in batch to reduce the number of drawing calls to the graphics card.

In the original ROAM implementation, the hierarchy starts with a rectangle which covers the whole area. To limit the depth, the terrain is divided into several patches with equal size. In order to remove crack between patches, each base triangle in the patch must have a pointer to its neighbour patch which must be either of the same level, one level finer, or one level coarser. The relationship between triangles is depicted in Figure 3.3.

Figure 3.3 Triangle relationship in ROAM[53]

Hierarchy is built on these patches. By using this method, the hierarchy depth is determined by the patch size. The patch size cannot be too small, yet not too large. A small patch will result in more triangles number for rendering. And a large patch will result in a deeper hierarchy level. Typically the patch size is 65, 129, or 257, depending on the terrain size.

The triangles generated from this computation are not transferred to the graphics card directly. Instead, they are stored in the vertex buffer for batch processing. The vertex buffer provides a mechanism to fill in vertex buffer data by the CPU, while GPU renders the previous batch of vertices at the same time. Hence, it provides a small degree of parallel processing in rendering.

Even though the patch implementation improves the terrain in many aspects, CPU computation is still expensive due to the construction of triangle

hierarchy for each patch. This problem leads to the implementation of tiled block terrain.

3.3 Tiled Block

Figure 3.4 Tiled Block in GeoMipMaps

Current GPUs are able to process millions of triangles at interactive frame rates. This important fact leads to a new category of terrain rendering: tiled block. In this algorithm, CPU workload is reduced to keep up with GPU's speed. On the other hand, GPUs are given more data to render. By using tiled block, numbers of primitive processing are reduced since the computation is executed for every patch instead of every triangle. Seamless Patches[54] and GeoMipMaps are some algorithms that belong to this category. In this project, GeoMipMaps is implemented as shown in Figure 3.4.

In order to provide data streaming and high quality rendering, some modifications are proposed as follows.

3.3.1 Data Structure

In the original algorithm proposed by de Boer[8], there are two levels of hierarchies, that is: the terrain and terrain patch. But, to support data streaming in rendering very large terrain, the terrain engine is modified such that it has three levels of hierarchies:

- **Terrain Patch.** As the bottom level of hierarchy, it is the basic unit used for rendering. So, instead of doing per-triangle computation, LOD is executed per-patch basis. In this class, neighbouring connection and minimum distance square for each LOD level are stored.
- **Terrain Block.** As the mid level of hierarchy, it is used to hold several terrain patches. This extra class is used to support large texture and data streaming by reducing system calls to hard disk to load a new data. Using terrain patch for streaming generates a lot of system calls because the data of terrain patch is typically very small.
- **A single terrain.** As the top level of hierarchy, it is used as an interface between a terrain and a game engine. It contains several terrain blocks. In this level, an algorithm decides if it should load new data from hard disk and release some data from memory, based on viewer position.

Figure 3.5 Active area on terrain block

Take a look at Figure 3.5. To prevent ‘unnecessary’ system call, the terrain defines an active area as an extension area of current block. This active area covers the entire current block area (block 13) and some parts of its neighbour (blocks 7, 8, 9, 12, 14, 17, 18, 19). This extension is used to delay the loading process since the camera might be going back to the previous block.

The terrain holds $m \times m$ blocks at one time. The m value is set based on the size of the terrain block. When the camera moves outside the active area, there will be a system call. The engine will invoke a thread to load some new blocks from the hard disk and release the others based on camera movement. To handle shared data for this thread, mutual exclusion is used. At the same time, it defines a new active area based on the new viewer position.

The computation of bounding volume and minimum distance square for each patch are expensive. In order to load the data as fast as possible, these values

are stored together with heightmap data. So, there is no computation during the loading process.

3.3.2 Rendering

In a frame update, the Quad-Tree is used to determine visible blocks and patches from the current viewer location. These visible blocks are then sorted based on their distances to the camera. The blocks near to the camera are rendered first. This method is used to speed up the rendering process by avoiding the same pixel to be processed more than once.

The algorithm computes the LOD level for each patch to construct the triangulation for the whole terrain. Rendering is executed by passing vertex and index buffer to rendering pipeline. To save GPU memory allocation, a small number of vertices are sent continuously. In the implementation, the GPU only needs to allocate 1000 vertex information. Since the terrain used data streaming, fog may be used to mask out the far away terrain area. Fog is implemented in shader by calculating the vertices distance to the camera.

Tiled Block solutions may produce discontinuity between neighbouring tiles when a patch has LOD level l and its neighbour has LOD level k , where $l \neq k$. So, it is necessary to solve this discontinuity in order to create seamless rendering. The original solution solves geometry discontinuity. However, visual artefact still appears as illustrated in Figure 3.6.

Figure 3.6 Visual artefact (red) due to wrong stitching

In order to solve this problem, an experiment is conducted by implementing the other proposed solutions. [1] modified the stitching triangulation by doubling the size of triangles adjacent to the larger patch (low detail) as illustrated in Figure 3.7. [37] modified it by providing the tessellation in low detail patch instead of high detail patch as illustrated in Figure 3.8.

Figure 3.7 Stitching proposed by Larsen[1]

Figure 3.8 Stitching proposed by Brodersen[37]

However, visual artefact also appears in these solutions since long and thin triangles are used in stitching process, as illustrated in Figure 3.9.

Figure 3.9 Visual artefact (red) due to wrong stitching

In order to remove the artefact completely, minimum angle constraint must be conformed. [55] has shown that the value of minimum angle for Delaunay triangulation must be 20° . Borrowing the concept from TIN, the proposed implementation limits the maximum LOD difference for each neighbouring patch to two using the Larsen stitching process.

3.3.3 Terrain Collision

The GeoMipMaps algorithm uses spatial partitioning data structure for rendering. For collision detection, the algorithm starts by finding the terrain block and patch that contain collision point P . Then, the level of detail for the patch is calculated by measuring the distance between the camera and centre of patch. This LOD value is used to compute the distance between each vertex in the corresponding patch so that the triangle that contains collision point P can be found. For the final step, the collision height is computed using barycentric coordinate.

In general, tiled block terrain must deal with stitching. So, the cost of finding the exact triangle is expensive since the neighbour information must be found. To reduce the computation, the algorithm could use an interpolation method, instead of barycentric coordinate. In this case, the algorithm must find four nearest vertices based on the LOD value.

Tiled block rendering reduces the CPU computation by processing patch instead of triangles. However, it is difficult to achieve frame-to-frame coherence since the triangulation for each patch depends on its neighbouring patches.

3.4 Nested Regular Grids

Figure 3.10 Nested Regular Grid in GeoClipMap

The GeoClipMap employs nested regular grids for terrain rendering. It is a simple and efficient method which is similar to texture image LOD. The

algorithm creates view-dependent terrain level of detail approximation. Hence, there is no computation for refinement. Figure 3.10 shows the GeoClipMap rendering output.

The implementation of GeoClipMap is similar to [9]. Some modifications are made to speed up rendering and to support GPU Terrain collision as follows.

3.4.1 Data Structure

There are two classes for this implementation: clipmap *level* and a single *terrain*. Clipmap *level* is used to store transformation values for the current level, such as level spacing, level translation, bounding box, and toroid origin for heightmap data. These values are computed in the initialization process. For the implementation of frustum culling and to reduce memory cost, the level is divided into some smaller blocks. As illustrated in Figure 3.11, there are four kinds of blocks based on their sizes: $m \times m$, $m \times 3$, $(2m + 1) \times 2$, and outer degenerate triangles.

Figure 3.11 Blocks in clipmap level[9]

A single *terrain* contains several terrain *levels*. Most of the data, such as vertex and index buffer, are stored in this class. One main difference from the original implementation is the storage of heightmap values as a texture. In the original implementation, each *level* stores a texture containing some portion of heightmap values. When the camera moves (translates), these textures values are updated. The levels which are near to the camera are updated more frequently than those which are far away. However, if the camera rotates, there is no need to update the elevation data on the textures.

To support GPU terrain collision, the elevation data is stored as single texture in the *terrain* class, instead of several textures in the *level* class. The textures can be combined by tiling them along 'X' direction, 'Y' direction, or both. If the tiling is set in both directions ('X' and 'Y'), then the new index for both X and Y values need to be re-computed to find the correct pixel coordinate on the texture given a level number. So, to access the index faster, the textures are merged in one direction only ('Y' direction), as shown in Figure 3.12.

Figure 3.12 Texture merging process

3.4.2 Rendering

The frame update starts by calculating the minimum *level* for rendering based on camera height. This step is necessary to prevent aliasing in rendering. Then, it will check whether the heightmap data needs to be updated based on camera position.

The blocks for each *level* are tested for visibility in frustum culling. The rendering starts from the *level* near to the camera. In order to speed up the rendering process, instance buffer is used to store transformation values for all visible blocks. Using this method, the number of *DrawPrimitive* calls per frame can be reduced.

3.4.3 Terrain Collision

Collision detection is implemented within the GPU as an extension of the rendering system. There are two purposes of this implementation: to handle a large number of queries and to avoid data duplication in CPU (since terrain data is stored in GPU). The reference on GPU programming can be found in [56].

The input is a 1D-array consisting of n collision points. If the input is transferred as Vertex Data, then they must be processed as *PointList*. In this case, GPUs treat the input as a collection of points for rendering. In the GPU processing, this input is transferred to Vertex Shader. However, since Vertex Shader does not do any computation, this input is just passed to Pixel Shader for collision detection computation.

There is one drawback in using Vertex Data: expensive Vertex Shader processing. In this method, there are n Vertex Shader computations to process n collision points. One way to reduce this is to use Textured Surface. The 1D-array of n collision points is transferred to GPUs as a 2D texture, as shown in Figure 3.13. To process this texture data, GPUs are given an instruction to draw a rectangle which covers the whole area of input texture data. So, there are only four Vertex Shader computations for four corners of rectangle. In the Pixel Shader, the texture data is fetched by using *tex2D* instruction.

Figure 3.13 1D arrays packed into 2D texture

There are only two kinds of output for the graphics pipeline: backbuffers for monitor display and texture render target. Between these two, the system can only use texture render target since the values can be fetched by CPU[57]. *GetRenderTargetData* is used to transfer back the result for CPU processing. In the processing setup, the GPU output is set to texture render target. Once the computation is done, the GPU output is set back to backbuffer.

3.5 GPU Optimizations

GPU computation offers several advantages such as a list of instruction set which cannot be found in CPUs instruction. This set of instructions is used to

support vector operation as most of GPU computations use vector unit. However, there are several drawbacks as well. GPUs generally do not perform well for dynamic branching. Besides, there is a setup delay to perform the computation in the GPUs. In the proposed implementation, shader is optimized as follow.

3.5.1 Branching

The graphics pipeline performs one operation for multiple sets of data (Single Instruction stream Multiple Data stream/SIMD). Starting from shader version 3, dynamic branching is possible as part of the instruction flow control. But, if there is a branching of if-then-else in shader instruction, the GPU processor needs to execute both branches[58]. Besides, there is an overhead for this branching instruction which may slow down the execution. In order to reduce these two overheads, the branching instruction is replaced by some mathematical formula.

```
if(A >= B)
    C = D
else
    C = D + 1.0

// the code above is similar to the code below:
float condition = step(A, B);
C = D + lerp(0, 1, condition);
```

3.5.2 Instruction-level parallelism

One of the main advantages of the GPU processor is vector arithmetic which is not available in CPUs programming[59]. In CPU implementation for collision detection, there are several similar computations for some scalar values, for example: alpha computation to compute morphing height. Since this computation is used several times, they can be combined into one instruction by using vector operations.

```
delta.x = (pos.x-translate.x)/2;  
delta.y = (pos.y-translate.y)/2;  
delta.z = (pos.z-translate.z)/2;  
delta.w = (pos.w-translate.w)/2;  
  
// the code above is equal with the code below:  
Delta = (pos-translate)/2;
```

Besides that, swizzle operator is used to create other vectors out of the existing ones. Basically vector data members are rearranged, duplicated, or omitted in any combination to create a new vector. In the GPU's implementation, a swizzle operator is used to compute new position after translation.

```
pos = translate.xyxy + pos.xyzy*multiplier;
```

3.5.3 Precomputation of loop invariants

There are several values which are uniform during kernel processing. These values are constant which are used by all fragments. Hence, they are best pre-computed on the CPUs and passed to GPUs as a uniform parameter. For

example, the fragment program requires the value of $k=1/(mNumLevel*mSize)$. It is possible to set parameters *mNumLevel* and *mSize* to be passed to GPUs and later on the GPU computes the value of *k*. But, to remove redundant computation, it is preferable to compute *k* on the CPU and pass the result to GPUs.

Chapter 4 Experiment and Analysis

This chapter presents the results of experiments on the terrain rendering algorithms with their analysis. First, experiments are conducted to compare the performance of current algorithms. Then, proposed implementations are compared with the current algorithms. Tables and figure are presented to show the strengths and weaknesses of each algorithm. The platform used for these experiments is a PC with processor Intel Core 2, 2.66 GHz, 2GB RAM, and graphics card nVidia GeForce 8600 GTS.

4.1 Algorithms Comparison

This experiment compares the frame rate, triangle number, and CPU time. The heightmap data is obtained from one commercial game engine. For ROAM and GeoMipMap, the rendering size is set to 4033 x 4033 and the patch size is set to 65. Since GeoClipMap uses nested grid for rendering, the rendering uses 6 levels and each level has a size of 127. So, the rendering area for GeoClipMap = $2^6 \times 127 = 4064 \times 4064$.

Algorithm	FPS	Average Number of Triangles	CPU Time (ms)
ROAM	54	123593	9.4
GeoMipMap	199	139759	2.63
GeoClipMap	466	83796	0.05

Table 4.1 Comparison of terrain rendering algorithms

Table 4.1 shows the comparison of several terrain rendering algorithms based on their data structures. The frame rates are calculated by taking the many samples from different view points for multiple frames. In general, the number of triangles changes when the view point changes. In ROAM, a triangle pool is used to allocate and de-allocate the triangles to the system. The number of triangles rendered is retrieved from the number of triangles used in the pool. In GeoMipMap, the number of triangles can be computed from the LOD levels for each visible terrain patch as well as the LOD difference with its neighbours. And in GeoClipMap, the number of triangles can be computed from the number of visible blocks in each level.

As shown in the Table 4.1, ROAM has lowest frame rate due to expensive triangle computation in the CPU. One advantage of this algorithm is that the maximum number of triangles can be set by modifying the value of variance during run time. However, popping effect appears more visible using this algorithm, especially in the early execution of rendering as shown in Figure 4.1 and Figure 4.2.

Figure 4.1 Initial rendering of ROAM in wireframe mode

Figure 4.2 Initial rendering of ROAM in wireframe mode at another instant

Figure 4.1 and Figure 4.2 show terrain rendering using ROAM algorithm on the same surface but at different time. These figures are captured during the early execution of the ROAM algorithm without moving the camera position. As shown in these figures, popping effect appears more visible since the algorithm imposes the limit of triangle number. Hence, it will stop splitting when the limit has been reached and it will adjust the variance value for the next frame accordingly.

Initially, ROAM generates triangulation as shown in Figure 4.1 which has sparse triangles near to the camera since the splitting process starts from the corner of terrain which is quite far from the current position. This splitting process stops before it reaches the area near to the camera since the maximum triangle number has been attained. For the next frame, the variance value is updated to generate better triangulation such that it has more triangles for the area near to the camera, as shown in Figure 4.2.

To reduce this popping effect, the algorithm can be improved by using a priority queue which stores list of patches to be split based on camera distance. However, this implementation will reduce the frame rate since it has to sort all the patches for every frame. Besides, the exact variance value must be known beforehand.

As shown in Table 4.1, although GeoMipMap has more triangles than ROAM's, the frame rate is much higher since the triangulation is computed for each patch instead of individual triangles. However, the current implementation has flaws in the stitching process such that visual artefact appears in certain area.

Besides, data streaming takes much time since it needs to compute bounding box and minimum distance square for each patch.

GeoClipMap performs the best among the others since it has the highest frame rate and lowest CPU Time. However, the algorithm has varying performances, depending on the level sizes and number of levels. When the level size is set to 255, the level number is set to 5, and the rendering area is $2^5 \times 256 = 4080 \times 4080$, the frame rate drops down to 50% compared to the previous settings. It is mainly due to more triangles that must be rendered. Besides that, the performance of GeoClipMap also depends on the camera movement. When the camera is not moving, the frame rate increases about 20% compared to the frame rate when it is moving. The frame rate drops mainly due to the texture data transfer between CPU and GPU memory during modification. As the texture size gets larger, more time is required for transferring the data. The current implementation of GeoClipMap can still be improved to support GPU terrain collision to handle large number of queries.

4.2 Proposed Implementation

The experiments are conducted to compare the performance of the proposed implementation. There are three modifications proposed: data streaming, stitching, and GPU terrain collision. The first two modifications are applied for the GeoMipMap algorithm. And the last one is applied for the GeoClipMap algorithm.

4.2.1 Data Streaming

This experiment compares the performance of data streaming between the original implementation and the modified one. Both implementation has patch size 65. The original implementation does not have terrain block and the number of patch used for rendering = 55×55 . The modified implementation uses terrain block with size = 513 and the number of block = 7×7 . The experiment is conducted by moving the camera in straight direction with time limit = 30 second.

	CPU Time (s)	# data streaming
Two Levels	1.5	23
Three Levels	0.06	3

Table 4.2 Data streaming time comparison

Table 4.2 shows the comparison of data streaming times between the original implementation (two levels) and the modified one (three levels). Three levels structure generate less data streaming invocation compared to two levels structure. In general, many data streaming invocations generates overhead in the application. If the application only renders the terrain, this overhead is not visible to user. However, if the terrain is implemented as part of a game engine, then data streaming could cause lagging in rendering process.

Besides, three levels structure take less time for loading the data since the data has been pre-computed. In this experiment, the computation of bounding volume and minimum distance square values have been proven to be expensive. So, it is better to pre-compute them and store the values as part of

the data. These extra data only requires a little more space in hard disk. For patch size = 65, the extra data contains bounding box (6 floats) + minimum distance square values ($(\lceil \log 64 \rceil) = 6$ floats). In total, they requires 12 float data only.

4.2.2 Stitching

	FPS	Triangle Number
Old Stitching	320	81934
New Stitching	302	85321

Table 4.3 Stitching comparison

This experiment compares the performance of stitching method between the original implementation and the modified one. Both implementation has block size = 513, number of block = 7 x 7, and patch size = 65.

Table 4.3 shows the comparison between the old implementation of stitching and the new one. The new implementation generates only 4% more triangles and the frame rate drops by only 5%. This little disadvantage can be compromised in order to create good terrain visualization since the new implementation generates triangulation which is free of visual artefact.

Figure 4.3 Visual artefact in GeoMipMap

Figure 4.4 Original stitching in GeoMipMap

Figure 4.3 and Figure 4.4 shows the original implementation of the GeoMipMap algorithm which has visual artefact in certain areas due to wrong stitching. It can be fixed by modifying the triangulation and limits the maximum LOD difference for each neighbouring patch to two to avoid thin triangle as shown in Figure 4.5 and Figure 4.6. Other examples can be found in the appendix.

Figure 4.5 No visual artefact in new implementation

Figure 4.6 New stitching

4.2.3 Collision Time

The experiment is conducted by making 1000 queries of object moving in certain direction. The direction is decided by a random number. During initialization, the object is distributed uniformly on the terrain surface. This query is executed for each terrain rendering algorithm. To obtain a fair result, the query is executed for 10000 frames.

Algorithm	Time
GeoMipMap	6.25
ROAM	7.15
GeoClipMap	10.13

Table 4.4 Collision time comparison

Table 4.4 shows the collision time for various algorithms in second for 10000 frames. GeoMipMap algorithm achieves the best result. The flows of the algorithm for GeoMipMap collision detection is shown in Figure 4.7.

Figure 4.7 Terrain collision algorithm for tiled block terrain

The unexpected result comes from ROAM implementation. Although the complexity depends on the hierarchy depth, the computation time is less than GeoClipMap algorithm. It happens because the level of ROAM patch is not too deep once level of detail is applied. Besides, GeoClipMap needs to compute the morphing height value which is quite expensive.

The time required for GeoClipMap is about twice of TIN. However, it can be improved by implementing terrain collision in GPU.

Figure 4.8 Collision time comparison of CPU and GPU implementations

Figure 4.8 shows the GeoClipMap collision time comparison between CPU and GPU implementations. The GPU implementation must transfer the query of collision point coordinates from CPU to GPU. Once the GPU finished the computation, the result has to be transferred back from GPU to CPU. The time measurement for GPU includes the time required to copy the collision points to texture and fetch back the result from GPU render target texture to CPU main memory. This texture data transfer will slow down the performance of GPU collision detection. Before the time measurement starts, GPU is queried if it is idle. Once the GPU sends a signal that it is idle, the computation can then be started.

When the number of queries is small, CPU performs better. It happens due to the setup cost for GPU computation and data transfer between CPU and GPU. When the number of queries equals to 130, the equilibrium point is met. At this point, both CPU and GPU have the same computation time. As the query number increases, the GPU implementation performs better. The GPU performance is almost unaffected by the number of queries since the increment is very slow.

4.3 Terrain Collision Detection Optimization

The computational cost for terrain collision detection not only depends on the algorithmic complexity, but also the number of queries/tests required. Hence, it is important to reduce the number of queries by applying the test only to the places where a collision could occur. Some methods have been developed such as: object categorization and time step.

There are many objects in a terrain. In general, terrain collision detection is only applied to the moving object. These moving objects can be grouped into three categories.

The first category is a group of objects which is always on the terrain surface. So, collision points are defined only at the object surfaces which collide with the terrain. For example, for human and animal, the collision points are the foot. And for vehicle, the collision points are the tyres.

The second category is a group of objects which never touch the terrain surface. These objects always maintain a minimum distance to the terrain. Hence, collision point can be reduced up to one point only depending on the size of object. Birds and airplanes can be grouped into this category.

The last category is a group of objects which collide with the terrain only for certain time. In other words, the object can be either on the terrain surface or in the air. The main difference with the other two categories is that the object does not have any information which part of the model surface that may collide with the terrain. So, the object in this category usually requires many collision points to obtain the contact point. The number of collision points depends on the model complexity. To reduce the number, these points are only placed on the potential surface which may collide with the terrain.

By grouping game objects into several categories, the number of queries can be reduced. It is sometime necessary for an object to switch from one category to another. For example, an airplane before taking-off belongs to the first

category, but when it is in the air, it belongs to the second category. And when this airplane crashes, it belongs to the third category, since the contact point must be obtained.

An experiment is conducted by making 1200 objects which can be categorized into three groups as explained before. Each group has 400 objects. To make it simple, some assumptions are made. If no object categorization is employed, then all objects have eight collision points since they use bounding box. If the objects are categorized, then the first group only has four collision points for each object (assuming that they are cars which has four wheels); the second group only has one collision points (assuming that they are birds); and the last group only has eight collision points (assuming that they are boxes). GeoMipMap terrain rendering is used for this experiment. To obtain a fair result, the query is executed for 10000 frames.

	Total number of queries per frame	Total Time
No object categorization	9600	62.08
With object categorization	5200	32.26

Table 4.5 Collision time comparison using object categorization

Table 4.5 shows the result of the experiment. When no object categorization is employed, the total number of queries equals to the total number of object multiplied by the number of collision points = $1200 \times 8 = 9600$. This number is about twice of the new implementation with object categorization. As a

result, the previous implementation takes more computation time compared to the new one.

The second optimization is time step. Basically, the method is to compute terrain collision for certain period of time only. So, the collision for some moving object is not computed for each frame. For example, a bird model maintains certain distance from the terrain surface. Once the bird's ideal position is achieved, the bird can keep moving without computing the height collision for certain period of time. This time steps must be short to prevent unexpected effects.

Another experiment is conducted by making 1000 bird objects which maintain a relative distance from the terrain surface of 100. If the implementation does not use time step, then the collision is computed for every frame. And if the implementation uses time step, then the collision is computed for every 10 frames assuming the object will not collide with the terrain during this time interval. To obtain a fair result, the query is executed for 10000 frames.

	Total Time
No time step	6.25
With time step	0.72

Table 4.6 Collision time comparison using time step

Table 4.6 shows the collision time comparison using time step. When time step is employed, the total time required to compute the collision decreases.

Chapter 5 Summary

5.1 Summary

Nowadays, most research in computer graphics aims to produce photorealistic image which is computationally expensive. In game applications, aside from achieving high-quality images, there is a constraint in computation time in order to maintain their frame rate. This conflict of interest is the main concern in terrain rendering.

Dynamic terrain rendering can produce realistic terrain simulation and visualization. The drawback is the extra workload due to extra processing for triangulation to generate seamless terrain. However, geometry is not the only concern in terrain implementation. In a 3D game application, frame rate and visual quality are also important. Less number of triangles does not necessarily mean high frame rate. In some implementation, high frame rate can be achieved even though it renders more triangles than other algorithms.

In most cases, low CPU time means high frame rates. Low CPU time for terrain rendering is one of the desired traits since most elements in game also require CPU computation. To reduce CPU computation time, frame-to-frame coherence in terrain rendering is required so that the data update can be incremental.

5.2 Contribution

A number of innovations are presented to improve the terrain visualization. These innovations are useful in order to fully implement terrain in a game application.

5.2.1 Data Streaming

The first innovation of this research is the data streaming for tiled block terrain. As the terrain size gets larger, the data may not fit into the main memory. In this situation, paging is required to obtain terrain data from an external storage device at runtime. The main challenge is to overcome the slow transfer time of data from external hard disk to main memory.

Three levels structure and data pre-computation are used in order to produce smooth data streaming during rendering process.

5.2.2 Improved Stitching

Seamless geometry in tiled block terrain can be achieved by stitching. However, the current approach does not consider the triangle angles which may produce long and thin triangles during rendering. These triangles produce artefact in terrain visualization.

To improve the stitching process in tiled block terrain rendering, the maximum LOD difference between neighbouring patch must be enforced. The increase of triangles number due to this method is only about 6%. However, it can be compromised to remove visual artefact.

5.2.3 GPU Terrain Collision

For real-time computation, result accuracy can be sacrificed to gain speed. For terrain, the collision data may not necessarily equal to rendering data in order to simplify the calculation. Besides, rendering algorithm may need to be modified as well to improve the computation speed.

Even though terrain rendering algorithm can achieve higher frame rate, they may not do well if there are a lot of terrain collision queries. Most of the algorithms only concern with rendering, but not terrain collision. In a game application, terrain collision detection is necessary as part of game physics.

To support large number of terrain collision detections, GPUs can be used to perform the necessary computation. Frame-to-frame coherence (incremental update) is required to achieve this. Besides, the elevation data must be stored in the GPU for faster processing. Some rendering algorithms store the elevation data in multiple places with different memory allocation. For example, the original implementation of GeoClipMap stores the elevation data in many textures, depending on the number of levels.

To support GPU terrain collision, the elevation data must be stored in the GPU memory as one big texture whose value can be retrieved quickly without much extra computation. In general, GPU implementation performs well for large number of queries.

5.3 Future work

One main drawback of tiled block terrain rendering is due to the absence of frame-to-frame coherence. It happens because of the stitching process which constrains the patch triangulation such that it depends on its neighbour. If this stitching triangulation can be separated from the patch triangulation, incremental update might be achieved. Besides that, instancing method can also be used by providing the triangulation for each level of detail in the patch. If the heightmap data can be stored in GPU memory, then there is a possibility to implement GPU terrain collision. However, the organization for elevation data is required which requires extra effort.

It is also a good idea to explore TIN solution to render large terrain using discrete LOD. In particular, TIN offers static triangulation which does not require any CPU computation. The TIN terrain can be divided into several patches. Each patch contains different LOD of TIN data. The level is determined by computing the distance between the camera and the center of patch. Besides, data streaming can be achieved since the data have been divided into smaller one. One main requirement is to create seamless triangulation in the border of patches. However, this approach requires more memory which must be considered.

Chapter 6 Appendix

6.1 Visual Artefact in GeoMipMap

6.1.1 First Example

Figure 6.1 Original Rendering

Figure 6.2 Original stitching

Figure 6.3 Proposed rendering by Larsen

Figure 6.4 Proposed stitching by Larsen

Figure 6.5 New rendering

Figure 6.6 New stitching

6.1.1 Second Example

Figure 6.7 Original rendering

Figure 6.8 Original stitching

Figure 6.9 Proposed rendering by Larsen

Figure 6.10 Proposed stitching by Larsen

Figure 6.11 New rendering

Figure 6.12 New stitching

6.1.2 Third Example

Figure 6.13 Original rendering

Figure 6.14 Original stitching

Figure 6.15 Proposed rendering by Larsen

Figure 6.16 Proposed stitching by Larsen

Figure 6.17 New rendering

Figure 6.18 New Stitching

References

- [1] B. D. Larsen and N. J. Christensen, "Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail," *WSCG 2003*, vol. 11, 3-7 February 2003.
- [2] J. Andersson, "Frostbite Rendering Architecture and Real-time Procedural Shading & Texturing Technique," presented at the Game Developer Conference 2007, San Fransisco, 2007.
- [3] C. Ericson, *Real Time Collision Detection*. San Fransisco: Morgan Kaufmann, 2005.
- [4] D. Baraff, "Dynamic Simulation of Non-Penetrating Rigid Bodies," Doctoral Thesis, Computer Science Department, Cornell University, Ithaca, 1992.
- [5] P. Lindstrom, *et al.*, "Real-Time, Continuous Level of Detail Rendering of Height Fields," in *Proceedings of the 23rd annual conference on Computer Graphics and Interactive Techniques*, 1996.
- [6] M. Duchaineau, *et al.*, "ROAMing Terrain: Real-Time Optimally Adapting Meshes," in *Proceedings of the 8th conference on Visualization '97*, Phoenix, Arizona, United States, 1997.
- [7] F. Losasso and H. Hoppe, "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids," *ACM Trans. Graph.*, vol. 23, pp. 769-776, 2004.
- [8] W. H. d. Boer, "Fast Terrain Rendering Using Geometrical MipMapping," unpublished and only available at <http://www.flipcode.com/tutorials/geomipmaps.pdf>, 2000.
- [9] A. Asirvatham and H. Hoppe, "Terrain Rendering using GPU-based Geometry Clipmaps," in *GPU Gems 2*, M. Pharr, Ed., ed Taunton: Addison-Wesley, 2005, pp. 27-45.

References

- [10] A. Frühstück, "GPU based Clipmaps: Implementation of Geometry Clipmaps for Terrain with non-planar basis," Master Thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, 2008.
- [11] L. J. Latecki, *et al.*, "Preserving Topology by a Digitization Process," *Journal of Mathematical Imaging and Vision*, vol. 8, pp. 131-159, 1998.
- [12] D. McQuail, *McQuail's Mass Communication Theory*, 4th ed. London: Sage, 2000.
- [13] M. Garland and P. S. Heckbert, "Fast Polygonal Approximation of Terrains and Height Fields," Carnegie Mellon University, Pittsburgh, Technical Report CMU-CS-95-181, 19 September 1995.
- [14] A. Saurabh and R. Maheshwari, "Terrain Visualization using Real-time Optimally Adaptive Meshes," Indian Institute of Information Technology, Allahabad, Mini Project Report, 9 December 2003.
- [15] D. M. Mark, "Computer Analysis of Topography: A Comparison of Terrain Storage Methods," *Geografiska Annaler. Series A, Physical Geography*, vol. 57, pp. 179-188, 1975.
- [16] P. Su, "Efficient Parallel Algorithms for Closest Point Problems," Duke University, Technical Report DUKE--TR--1994--22, 1994.
- [17] P. Su and R. L. S. Drysdale, "A Comparison of Sequential Delaunay Triangulation Algorithms," in *Proceedings of the eleventh annual symposium on Computational Geometry*, Vancouver, 1995, pp. 61-70.
- [18] S. Raman, "Efficient Terrain Triangulation and Modification for Game Applications," Master Thesis, School of Computer Engineering, Nanyang Technological University, Singapore, 2008.
- [19] R. J. Fowler and J. J. Little, "Automatic Extraction of Irregular Network Digital Terrain Models," in *Proceedings of the 6th annual conference on Computer Graphics and Interactive Techniques*, Chicago, Illinois, United States, 1979.
- [20] M. McGuire and P. G. Sibley, "A Heightfield on an Isometric Grid," presented at the ACM SIGGRAPH 2004 Sketches, Los Angeles, California, 2004.

References

- [21] A. S. Aquilio, "A Framework for Dynamic Terrain with Application in Off-Road Ground Vehicle Simulations," Doctoral Thesis, Georgia State University, Atlanta, 2006.
- [22] X. Cai, *et al.*, "Real-Time Visualization of Dynamic Terrain in Off-Road Driving Simulation System," presented at the Proceedings of the Second Workshop on Digital Media and its Application in Museum & Heritage, 2007.
- [23] Y. He, "Real-Time Visualization of Dynamic Terrain for Ground Vehicle Simulation," Doctoral Thesis, The University of Iowa, 2000.
- [24] J. Olsen, "Realtime Procedural Terrain Generation," University of Southern Denmark, Funen, Technical Report, 31 October 2004.
- [25] K. Perlin, "Improving Noise," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, San Antonio, Texas, 2002.
- [26] F. Aurenhammer, "Voronoi Diagrams - a Survey of a Fundamental Geometric Data Structure," *ACM Computing Surveys*, vol. 23, pp. 345-405, 1991.
- [27] K. E. Hoff, *et al.*, "Fast Computation of Generalized Voronoi Diagrams using Graphics Hardware," presented at the Proceedings of the 26th annual conference on Computer Graphics and Interactive Techniques, 1999.
- [28] R. Pajarola and E. Gobbetti, "Survey of Semi-Regular Multiresolution Models for Interactive Terrain Rendering," *The Visual Computer: International Journal of Computer Graphics*, vol. 23, pp. 583-605, 2007.
- [29] R. Veguilla, *et al.*, "Issues in Terrain Visualization for Environmental Monitoring Applications," presented at the Fourth LACCEI International Latin American and Caribbean Conference for Engineering and Technology, Puerto Rico, 2006.
- [30] D. Luebke, *et al.*, *Level of Detail for 3D Graphics*. San Fransisco: Morgan Kaufman, 2003.
- [31] H. Hakl and L. V. Zijl, "Diamond Terrain Algorithm : Continuous Levels of Detail for Height Fields," *South African Computer Journal*, vol. 29, pp. 8-16, 2002.

References

- [32] J. Blow, "Terrain Rendering at High Levels of Detail," presented at the Game Developer Conference, San Jose, 2000.
- [33] M. Duchaineau. (2003, 20 August 2008). *ROAM Algorithm Version 2.0*. Available: http://www.cognigraph.com/ROAM_homepage/ROAM2/
- [34] T. Akenine-Möller, *et al.*, *Real-Time Rendering*, 3rd ed. Natick, Massachusetts: A K Peters Ltd, 2008.
- [35] D. Sanchez and C. Dalmau, *Core Techniques and Algorithms in Game Programming*. Indianapolis: New Riders, 2003.
- [36] D. Wagner, "Terrain Geomorphing in the Vertex Shader," in *Shader X2: Shader Programming Tips & Tricks with DirectX 9*, W. F. Engel, Ed., ed Plano, Texas: Worldwide Publishing, 2004.
- [37] A. Brodersen, "Real-Time Visualization of Large Textured Terrains," presented at the Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia, Dunedin, New Zealand, 2005.
- [38] B. Rupnik, "Rendering Large Terrains in Real-Time," presented at the Central European Seminar on Computer Graphics, Wien, 2007.
- [39] Y. Yuting and W. Guoping, "View-dependent Real-time Terrain Rendering Using Static LOD," presented at the International CAD Conferences and Exhibitions, Pattaya Beach, 2004.
- [40] C. C. Tanner, *et al.*, "The Clipmap: a Virtual Mipmap," in *Proceedings of the 25th annual conference on Computer Graphics and Interactive Techniques*, New York, 1998.
- [41] MSDN. (2008, 25 August 2008). *Direct3D Architecture (Direct3D 9)*. Available: [http://msdn.microsoft.com/en-us/library/bb219679\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb219679(VS.85).aspx)
- [42] J. Andersson, "Terrain Rendering in Frostbite using Procedural Shader Splatting," presented at the ACM SIGGRAPH 2007, San Diego, California, 2007.

References

- [43] J. Owens, *et al.*, "A Survey of General-Purpose Computation on Graphics Hardware," presented at the Eurographics, Dublin, 2005.
- [44] J. Owens, "Streaming Architectures and Technology Trends," in *GPU Gems 2*, M. Pharr, Ed., ed Taunton: Addison-Wesley, 2005, pp. 457-470.
- [45] M. C. Miller, "Multiscale Compression of Digital Terrain Data to Meet Real-Time Rendering Rate Constraints," Doctoral Thesis, University of California, Davis, 1995.
- [46] S. Zerbst and O. Duvel, *3D Game Engine Programming*. Boston: Thomson Course Technology, 2004.
- [47] A. T. Lake and A. S. Kalra, "Grid-based Loose Octree for Spatial Partitioning," United States Patent 7002571, 21 February, 2006.
- [48] J. S. Hofmann, "The Render Graph: A Data Structure to Aid in the Interactive Display of Scene Graph Data," Master Thesis, School of Engineering and Applied Science, The George Washington University, Washington, 2003.
- [49] J. Zhang and G. Chen, "Exploiting Frame-to-Frame Coherence for Rendering Terrain Using Continuous LOD," in *Advances in Artificial Reality and Tele-Existence*, ed Berlin: Springer, 2006.
- [50] D. Conger, *Physics Modeling for Game Programmers*. Boston: Thomson Course Technology, 2004.
- [51] P. Jimenez, *et al.*, "3D Collision Detection: a Survey," *Computers and Graphics*, vol. 25, pp. 269-285, 2001.
- [52] P. Melchart, "Terrain Following and Collision Detection," presented at the Central European Seminar on Computer Graphics for students, Bratislava and Vienna, 1997.
- [53] B. Turner. (2000, 30 May 2009). *Real-Time Dynamic Level of Detail Terrain Rendering with ROAM*. Available: http://www.gamasutra.com/view/feature/3188/realtime_dynamic_level_of_detail.php

References

- [54] Y. Livny, *et al.*, "Seamless Patches for GPU-based Terrain Rendering," *The Visual Computer: International Journal of Computer Graphics*, vol. 25, pp. 197-208, 2009.
- [55] J. Ruppert, "A Delaunay Refinement Algorithm for Quality 2-Dimensional Mesh Generation," *Journal of Algorithms*, vol. 18, pp. 548-585, 1995.
- [56] D. Göddeke. (2005, 13 January 2009). GPGPU--Basic Math Tutorial. Available: <http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu>
- [57] M. Harris, "Mapping Computational Concepts to GPUs," in *GPU Gems 2*, M. Pharr, Ed., ed Taunton: Addison-Wesley, 2005, pp. 457-470.
- [58] E. Kilgariff and R. Fernando, "The GeForce 6 Series GPU Architecture," in *GPU Gems 2*, M. Pharr, Ed., ed Taunton: Addison-Wesley, 2005, pp. 471-491.
- [59] C. Woolley, "GPU Program Optimization," in *GPU Gems 2*, M. Pharr, Ed., ed Taunton: Addison-Wesley, 2005, pp. 471-491.