Real-time Terrain Rendering and Processing

A Report submitted in partial fulfillment of the
requirements for the degree of

Master of Science (by Research)
in
Computer Science and Engineering

by
Shiben Bhattacharjee
200607022

International Institute of Information Technology
Hyderabad, INDIA
July 2010

Real-time Terrain Rendering and Processing

A Report submitted in partial fulfillment of the
requirements for the degree of

Master of Science (by Research)
in
Computer Science and Engineering

by

Shiben Bhattacharjee
200607022
shiben@research.iiit.ac.in

International Institute of Information Technology
Hyderabad, INDIA
July 2010

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled “Real-time Terrain Rendering and Processing” by Shiben Bhattacharjee, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Advisor: P. J. Narayanan

To my Mom, Dad and Sister

Acknowledgements

I would like to thank my advisor, Prof. P. J. Narayanan, who have been very patient, motivating and have helped in many ways to tackle different problems. I have learnt a lot from him and I thank him for being very supportive and understanding.

Most importantly, I thank my Mom and Dad for so easily supporting me with everything I wish to go for, they are very nice people. My sister and my brother-in-law *(jijaji)* as well have been really nice to me and I wish them all the good blessings in this world.

I thank the masters program for giving me an opportunity to excel and be in a position to think about a perspective in life. I'll thank the CVIT research lab and the faculty for giving an awesome workspace and resources (graphics cards, w00t) to head start with.

I thank Soumyajit Deb who shared nice ideas at the early stage of my work. I also thank Suryakant Patidar, who was my project partner, for being a very good friend and helping me with everything. I thank my friends: *Dada, Bansee, Fanta, Parry, Shetty, Suhailer, Jagga, JJ, and Peetee*. Thanks for all the knowledge sharing and fun times! :D

I thank SONY for making the Playstation® (2, 3, Portable) and the killer games, especially the *God of War*® series. I also thank Dr. Gordon Freeman from Blackmesa to help me think about the most difficult problems like when to take cover, crouch, bunny-hop, break crates or solve puzzles. . . lol.

KTHXBYE

`4 8 15 16 23 42`

Abstract

Terrains are of great interest in flight simulators, geographic information systems and computer games. In computer graphics, terrain rendering is a special case because of their bulk. They cannot be handled as a single entity like other object models like teapots, cars and crates. Triangulated irregular networks of terrains are typically created by simplifying a dense representation. Such representations are popular in GIS and computational geometry. The recent trend in graphics is to use regular grid representations since they go well with today's graphics hardware. We explore different representation techniques to render terrains in this thesis. We look into real-time rendering, editing, and physical interaction with external objects on terrains. We also present a representation for efficient rendering of spherical terrains. Apart from rendering terrains realistically, we develop a method to render terrains artistically with painterly abstraction as well.

We create a system that exploits the power and flexibility of the modern GPUs to store, render, and manipulate terrains with minimal CPU involvement (CPU load $< 4\%$). The central idea is to use a regular-grid representation, hierarchically divided in fixed size blocks/tiles that change in resolution. The potentially visible portion of the terrain is cached at the highest necessary resolution and is rendered from the GPU. The cache is updated with the viewerpoint. Lower resolutions used for farther areas of the terrain can be constructed from the cache on the GPU. Todays GPUs have a limited capability to generate geometry within itself. Thus, the CPU can send a light geometry template which is expanded to the triangles by GPU. The CPU performs a coarse culling of the tiles with the GPU performing fine culling. Our system enables the terrain to be modified procedurally or edited interactively on the GPU with no CPU involvement. The terrain can also interact with a large number of external objects in real-time with all the physics calculations done on the GPU.

Terrains can also be mapped over a sphere for a planetary structure. However, terrains on sphere require a different representation due to the pole singularity of latitude-longitude representation. A 2D grid of height cannot be mapped directly on a sphere with uniform triangle count. Spheres can be rendered uniformly using Hierarchical Triangular Mesh (HTM) but the representation does not fit with 2D grid of heightmaps. We present a unified representation of HTM and clipmapping (flat terrain rendering technique) to render spherical terrains. Our representation works at any distance from the planet/sphere without any scripted work arounds.

The regular nature of terrain data also enables us to render the samples in a required order with no overhead of sorting. This introduces the possibility of applications which require sorted ordered

triangles. We explore non photo realisitic rendering of terrains. Artistic painterly appearance and the impression of terrains is created by effectively rendering several translucent brush strokes in a back to front order. The strokes are located in 3D space for frame-to-frame coherence during animation. The strokes are oriented along the slope of terrain analogous to the way artists paint on canvas. We use shaders to render strokes in real-time. A level of detail scheme is used to maintain a uniform stroke density in screen space. Various styles can be achieved with different stroke variations. We achieve real-time painterly rendering with a combination of object space positioning and image space rendering of strokes.

Contents

List of Figures

Chapter 1

Introduction

Terrain refers to the geometric structure of a land surface and is a physical feature for many scientific and engineering study. Topography, relief, tract and elevation are used as synonyms. Terrains have been studied for long because its understanding is important for many of reasons. The terrain of a region determines its envrionmental conditions which relate to fertility of land and water movement, and thus agriculture and hydrology. Weather patterns are greatly effected by the terrain. Terrains also determine the suitability for human settlement; flat, fertile land, river banks and plateaus tend to be better populated. Transportation is mostly dependent on terrains. Terrains are given critical importance in military operations mostly for movement of vehicles, placement of artilleries for best range, strategic positioning of armed forces etc. The structure of terrains has been exploited in wars for both defensive and offensive modes of operations. The importance of terrain extends to its formation as well. The topology of a region can be due to volcanic erruptions, earthquakes, faults, landslides, water movement or meteorite impact.

An important aspect of the study of terrains involve terrain visualization. Traditionally, for the purpose of visualization, maps have been used extensively. Topographic maps represent the shape or form of the land surface. The maps represent 3D surface as a two dimensional entity preserving the surface structure as much as possible. Maps are mostly used as a basis for visualizing landscape morphology but it is expected that users will be interested in understanding the shape of the surface and may wish to measure distances, slopes or heights. Thus, a high degree of accuracy is required. This creates a conflict between the requirements for metric accuracy and visual realism. Mapping techniques such as isolines provide an accurate representation of landforms in a way that is planimetrically correct but is not well suited to its visualization. The best topographic maps combine contours with relief shading or layer tints to provide visually more effective representation of terrain such as physical maps of an atlas. Topographic maps are usually created by sampling survey points having known elevations. These may be the only truly accurate elevations on the map.

With computer graphics, it is possible to visualize terrains in their original form in three dimensions unlike topographic maps. Computer graphics is the study of digitally synthesizing objects (rendering) and manipulating geometric information for appropriate visual content. Terrains can be rendered by

computers on a visual output device to a high degree of accuracy for visualization. Virtual terrains have a lot of uses in land planning and usage, urban planning, visualization of weather and other environmental attributes, planning for strategy and tactics of military operations, transportation, virtual tourism and travel planning, virtual Bathemetry for submarines, Design of radio/TV/cellular transmitter placement and signal analysis, Education, geographical & general reference, Games and entertainment, Real estate etc.

Virtual terrains clearly have many applications and the list keeps growing. To render terrains, a terrain in the form of data (i.e. digital form) is needed as input to the renderer application. The most basic type of representation of terrains in digital form is a height-map or *digital elevation model (DEM)*. A DEM is a representation of the topography of the Earth or another surface in digital format. In contrast with topographical maps, the height information is stored in a raster format. That is, the map will divide the area into rectangular pixels and store the elevation of each pixel. In that sense, digital elevation model (DEM) data are sampled arrays of surface elevations in raster form. A DEM is same as a grayscale image with each pixel representing an elevation value (Figure 1.1). Since such data is raw, DEMs are

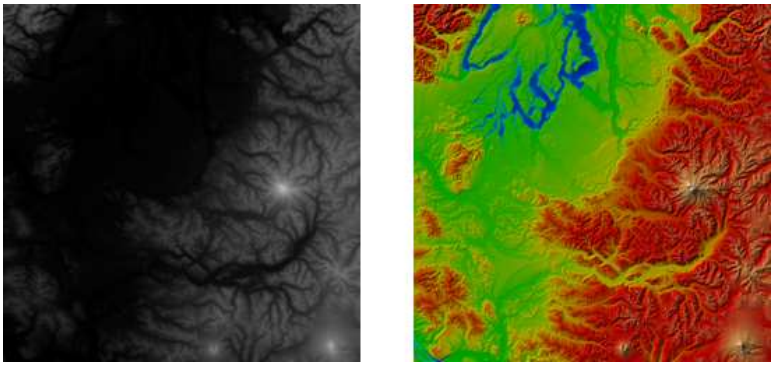

Figure 1.1 Left: Elevation data of Puget Sound. Right: Color texture for the terrain. *(Dataset Courtesy: The United States Geological Survey (USGS), made available by The University of Washington)*

also used to extract terrain parameters, model water flow or mass movement other than visualization as 3D rendering. Digital elevation models may be prepared in a number of ways, but they are frequently obtained by remote sensing rather than direct survey. One technique for generating digital elevation models is interferometric synthetic aperture radar; two passes of a radar satellite (such as RADARSAT1) suffice to generate a digital elevation map tens of kilometers on a side with a resolution of around ten meters. One also obtains an image of the surface cover. Other methods of generating DEMs often involve interpolating digital contour maps that may have been produced by direct survey of the land surface; this method is still used in mountain areas, where interferometry is not always satisfactory. Note that the contour data or any other sampled elevation datasets (by GPS or ground survey) are not DEMs. A DEM implies that elevation is available continuously at each location in the study area. The quality of a DEM is a measure of how accurate elevation is at each pixel (absolute accuracy) and how accurately is the morphology presented (relative accuracy). Several factors play important roles

for quality of DEM derived products: terrain roughness, sampling density (elevation data collection method), grid resolution or pixel size, interpolation algorithm and vertical resolution.

In computer graphics, objects to be rendered should be representated in a triangle mesh form. A heightmap contains only the elevation values, but because of its regular grid nature, the spatial ground coordinates are straight forward to calculate from the index of the height value. The connectivity is implied in the mesh structure. Thus heightmaps are sufficient to render terrains. The main difference between a terrain and any other model is that, though terrain is a single model in the graphics system, part of it is viewed from close and part of it is seen from far, at the same time, at every view point. Whereas other models (e.g. rocks, cars, chairs, marines, guns, aliens) are either viewed from far or from near on a whole. Bigger models (e.g. castles) are often broken in smaller models which are handled independently. The detail of these models is changed according to its distance from the camera. Distance of an object from the camera is a rough measure of how much area the object will cover on the screen (farther objects occupy less screen space, closer objects occupy more). The different *levels of detail* of a main model are kept separately as different models altogether and are selected to render given its distance from the camera. This is not easily possible for terrains since terrains are continous surfaces and any discontinuity due to level of detail management ruins the visualization experience. Terrain can become one of the most critical components in the scene that is being rendered. To keep the terrain engine running in real time can be a difficult task. To be effective, the terrain needs to meet a number of requirements, many of which can compete with each other. A terrain should appear to be continuous to the end user, yet the mesh should be simplified or culled where possible, to reduce the load on the graphics hardware. In a gaming system, for example, terrain is rendered till the point a player can reach, and then use a terrain drawn onto an imposter to simulate hills or mountains in the distance. The terrain should appear realistic to the setting for the environment, yet this can be taxing on the video card, and a balance needs to be maintained.

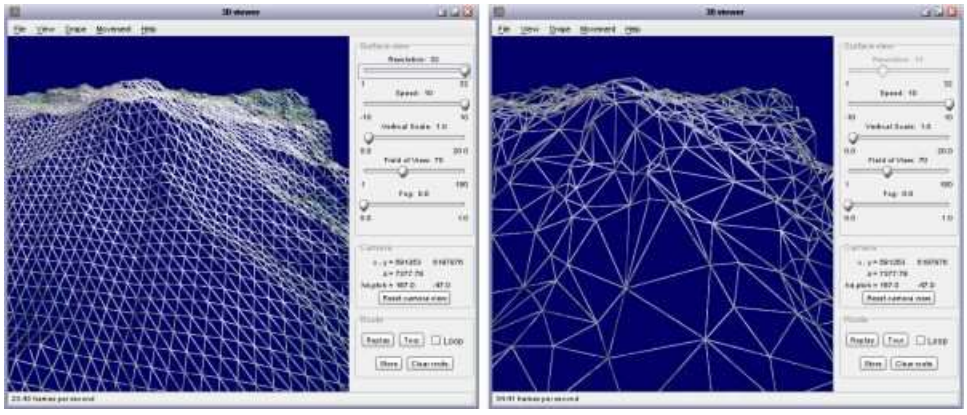

Figure 1.2 Left: Terrain with regular grid. Right: Same terrain with TIN. *(Image Courtesy: LandSerf www.landserf.org, User's Guide)*

Terrain in many regions can be flat, because of which the DEM data redundantly contains same elevation values over a long range of indices. Since the data is regular, there is no adaptive control over the detail available in a 2D grid of heightmaps, it is all at the same detail. To reduce this redundancy, terrains are sometimes converted from a regular grid of heights to a triangulated irregular network (TIN). The irregular representations contain fewer triangles but are more complex to represent and manipulate (Figure 1.2). They provide rendering efficiency at the cost of ease of performing other operations like editing of the terrain or physics computations over it. Smooth level of detail is hard to maintain with TIN based terrain rendering. TINs have been used to render terrains in games and other applications for a long time, but with the introduction of new programmable GPUs, DEM data tend to be better since they can be treated as textures (images) in the GPU memory. The grid of heights is usually split into smaller rectangular blocks for efficient storage and manipulation. Such a grid can be stored on the video memory of the GPU and rendered directly from there.

Terrain rendering goes beyond rendering terrains over a flat base. Many applications like to show a fast fly through over the terrain, from ground level to great heights from which the terrain of the planet is seen. A 2D grid seems to correspond directly with a latitude and longitude based representation of spherical terrains, but this does not distribute the samples regularly at all latitudes and has singularity at the poles (Figure 1.3(a)). Spherical terrains have the additional difficulty of handling regular sampling of

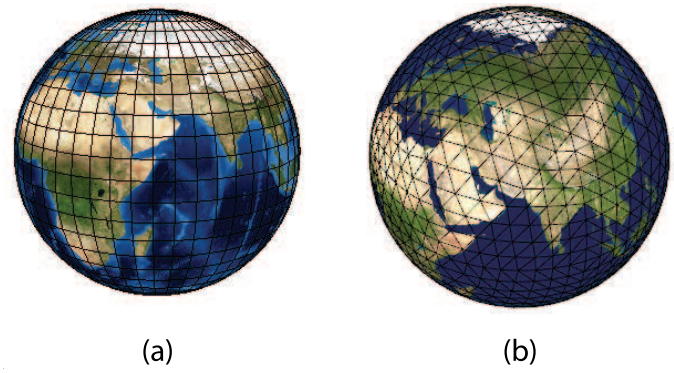

Figure 1.3 (a) Cartography system does not distribute samples equally all over the sphere. (b) Hierarchical Triangular Mesh is a technique which represents a sphere uniformly.

heights, time consuming spherical coordinates calculations, better memory management for the planet and lack of precision in calculations of coordinates due to the very large number of samples. Few graphics applications have been able to show a seamless fly through from space to surface of planet (or vice versa). Most games use camera tricks and scripted sequences to give an unfinished experience. Google Earth^{TM} seems to do the job but has severe pole singularity problems and lacks a smooth level of detail scheme. A good representation which can sample a sphere regularly and still allowing the use 2D grid of heights to render the mapped terrain is thus necessary to handle terrains on spherical planets (Figure 1.3(b)).

Figure 1.4 Left: Real Photograph, Right: Painterly Rendering of the photograph. *(Image Courtesy: A. Hertzmann [23])*

Terrains are sometimes also required to be visualized in an artistic form. The intentions of an artist come out as the aesthetics and expressiveness of the painting. The accurate rendering done by computers fails to provide images with a such a feeling. Animations are therefore often created by artists by painting a number of frames and is a tedious job. Computers have been used over the years to generate the surrounding environments of the main characters. This reduces the artist's effort, but leads to a visual disparity between the hand drawn objects and the environment. Painterly rendering, a *non-photo-realistic rendering* technique, can bring artistic abstraction to a real photograph (Figure 1.4) or a rendering and thus mix the computer generated scenes with the hand drawn elements. Therefore, painterly rendering has attracted the attention of graphics researchers. Creating abstraction of landscape and terrains seems an interesting problem since they are common in artistic creations and animations. Painterly rendering technique for general polygons exists. These cannot be applied directly to terrains because of level of detail complexities and richness due to long view range as explained before. An optimal composition of terrain rendering methods and painterly rendering is essential for real-time performance and high quality rendering.

In this thesis, we present a real-time terrain rendering system which is capable of producing high quality outputs with efficient rendering speed. The thesis also shows how our system allows manipulation of terrain, particularly interactive editing of terrain, deformation over time and objects behaving according to physics over the terrain. We explore rendering terrains over spherical structure for planets with uniform sample distribution of samples (heights) on the surface of the planet (sphere) as well. We also used our flat terrain rendering system to created non-photorealisitc illustrations of terrains which look like paintings. A list of contribution of the thesis is compiled in the next section.

This thesis is categorized in a number of Chapters. Chapter 2 sheds light on previous related work in the area of terrain rendering. It contains sections which are required for subsequent chapters. Chapter 3 explains the basic terrain system. In Chapter 4 we explain the implmentation details and some contributions at the side of exploiting the GPUs of our system. Chapter 5 focuses on a different problem of rendering terrains over a planetary structure and our approach. Chapter 6 explains how abstract illustration of terrains is done to mimic hand painted look. The following chapter counts the results and conclusions of the thesis.

1.1 Contributions of the thesis

1. A terrain representation that uses fixed-size blocks of grids and GPU caching. This enables fast rendering along with accessibility to edit and manipulate terrains, and simple physics operations.

2. A scheme of sending light geometry templates from the CPU to the GPU, which are expanded into the actual geometry. This keeps the CPU free to do other tasks while the GPU performs the bulk of the rendering work.

3. A hierarchical two-level culling scheme with the CPU culling in units of large tiles and the GPU culling in units of smaller tilelets for high rendering performance. The rendering rate doubles with this.

4. Clever interleaving of data transfer from the CPU to the GPU to keep the cache up- dated correctly without affecting the rendering rates. This guarantees frame-rates above a desired threshold all the time during rendering of arbitrarily large terrains.

5. Fast, interactive and procedural manipulation and editing of GPU-resident terrains using the fragment shader. The highly parallel GPU resources are employed protably to do this, improving the system performance.

6. A zero-overhead method to render samples of terrain in back to front order without the need of sorting. We utilize this to render alpha blended strokes in painterly rendering of terrains.

7. Level of detail scheme of terrains is used to reduce or increase detail of brush strokes according to distance in painterly rendering of terrains. The level of detail of the terrain is changed smoothly with distance from the viewpoint. This avoids the problem of strokes getting cluttered at far distances, which can be visually distracting. Level of detail also reduces the rendering load.

8. Hexagonal Geometry Clipmaps for spherical terrain rendering which is a simple combination of Geometry Clipmaps (an efficient terrain rendering technique) and Hierarchical Triangular Mesh (A uniform sampling and indexing algorithm for spheres)

9. A new method to sample maps of planet earth which has no redundant information compared to the conventional cartographical maps we see in atlas. We save 50% of storage space with this representation and is required for the terrain data to fit with hexagonal geometry clipmaps.

Chapter 2

Previous Related Work

In this chapter, we review the related prior work on terrain representation and rendering. We discuss the previous work on terrain rendering in general in Section 2.1. In Section 2.2, we describe major earlier work in spherical terrain rendering. In the following section, we look at a brief history of non photo-realistic rendering, and its relation with terrain rendering.

2.1 Terrain Rendering and Manipulation

Terrain rendering is a well-studied problem. Triangulated irregular networks are created from regular grids with connectivity typically decided using a triangulation process such as Delaunay's [10, 13, 15, 16, 28]. They provide rendering efficiency at the cost of ease of performing other operations and are popular in GIS and computational geometry, but the recent trend in graphics is to use regular grid representations. Hierarchical tree structures like quadtrees exploit the 2D grid structure of terrains and have been used to represent them. They are storage efficient and lend themselves well for compression [6, 14, 35, 36, 42], but not always efficient for random access of the height values. The hierarchical approach has also been combined with triangulation for better rendering performance [8, 33]. Terrains have also been partitioned into fixed size square patches of different resolutions. The tiled structures provide compact representation and easy rendering. The block boundaries can show artifacts which are taken care of using special zero-area triangles and stitching [38, 49]. The fixed size blocks also limit the range of resolutions supported when a tile is reduced to a single height.

Losasso and Hoppe introduced a multiresolution, fixed memory size scheme for efficient representation and rendering of large terrains, called the geometry clipmaps [38]. They use a square region around the viewer as a geometry clipmap with high resolution at the centre and lower resolutions on the outer rings. The fixed memory structure involves constant rendering load. The geometry clipmap were stored in the GPU and rendered from there [2] and were also extended to spherical coordinates [11]. Geometry clipmaps provide good rendering performance, but the representation does not lend itself to editing or modification of the terrain, which is possible especially on today's GPUs. They also store multiple resolutions of portions of the terrain. Schneider and Westermann report an LoD based rendering technique

for terrains that use the GPU extensively using a multiresolution, tiled structure [46]. Their method gives high rendering rates but has all limitations of the fixed tile representations, such as hard boundaries that need special handling. Rectangular patches have been used as seamless patches with stitching strips but requires the simultaneous storage of multiple resolutions [37].

Terrains are traditionally considered static and fixed. Deforming and editing are performed rarely during visualization. Earlier work on terrain modification include multiresolution detail patches by He et al. [22] and modelling soil slippage by Li and Moshell [34]. The height-maps are amenable to quick editing, unlike the irregular representations. Atlan and Garland edit the terrain in real-time using a few editing strokes for applications such as geological simulations [3], using a wavelet-based representation. They use a two-step approach to recover the terrain and to edit it. Schneider et al. present a system to generate very large landscapes on the GPU using projected grids [45]. They generate and render procedural terrains using shaders but do not handle real terrains. Physics of particles has always been studied as an independent problem. Recently Kipfer at al. [31] and Kolb et al. [32] implemented a particle system on the GPU.

2.2 Spherical Terrain Rendering

When terrains are mapped over a planetary structure, most commonly a sphere, we call it *spherical terrain rendering* in short. Spherical Terrain Rendering involves additional problems involving correct distribution of samples over the surface of sphere, representation of sphere, level of detail, view range, spherical surface coordinates and calculation overheads etc.. Early work in Spherical Terrain Rendering originate from plannar terrain methods and then their extentions to a spherical base.

O'Neil [41] and Gerstner [18] extended the ROAM algorithm [14] for a planetary structure. The problem with above and other non-grid based algorithm is that the data needs real-time processing. Addressing this issue, Hill [27] favored a tiled block solution. Cigoni et al. [8] extended the BDAM algorithm to suit planets [9]. These algorithms divide the planet in square regions and use a cube as the base. Google Earth, a successful commercial application, suffers from flickering, possibly due to pole singularity problems. Modern GPU friendly Geometry Clipmaps [2] was first used by Clasen and Hege [11] in spherical clipmaps for fast rendering, however, lack of direct correspondence of height data to vertices produced aliasing issues.

Independent from the research in Spherical Terrain Rendering, Hierarchical Triangular Mesh [48] caught our interest which primarily focuses on supporting geospatial indexing and searching at different resolutions, from arc seconds to hemispheres. We are more interested in HTM's subdivision and its uniform sampling of the sphere. Modern GPU friendly terrain rendering techniques, however, rely on right triangles fundamentally for the tessellation of the terrain. HTMs are strictly based on equialateral triangles. To use any existing terrain rendering method, say Clipmaps [2], we have to find a way of making that technique work with equilateral triangles. Our goal was to use the best of sphere representation

systems (HTM) and efficient terrain rendering methods (Geometry Clipmaps [38]), and create a simple combination of two, yet technically sound.

2.3 Non-Photo Realistic Rendering and terrains

Abstract representation of still images was introduced by Haeberli [19] using image color gradient and user interactivity for painting. Hertzmann [23] places curved brush strokes of multiple sizes on images for painterly rendering. The technique fills color by using big strokes in the middle of a region and uses progressively smaller strokes as one approaches the edges of the region. Shiraishi and Yamaguchi [47] improves the performance of above method by approximating the continuous strokes by placement of rectangular strokes discreetly along the edges to create painterly appearance. Santella and DeCarlo [44] uses eye tracking data to get points of focus on images and create painterly rendering with focus information. All these techniques work well on single images but involve iterative techniques that make them cumbersome for real-time applications [25]. Also if they are applied on each frame of an animation independently, it can lead to a flickering of strokes due to incoherence of strokes between frames. Painterly rendering has been tried and made coherent on videos as well [26], [21], but these techniques are not well suited for 3D rendering.

Painterly rendering for animation was introduced by Meier [40]. She eliminated shower door effect and achieved frame to frame coherence by rendering several brush strokes whose positions stick the 3D model's surfaces. However, view dependent sorting of these strokes is required for alpha compositing, making it unsuitable for real-time animations. Recent work [4, 20] describe a real-time painterly process inspired by Meier using programmable graphics hardware. They render the polygonal model first and store the depth map. A second pass uses the depth map to remove occluded strokes so that the strokes/billboards can be rendered in any order. For a complex and distant scene, such as a terrain, the inaccuracies due to precision in the depth map and comparison at boundaries can reduce the visual quality. Terrains are rich models containing many samples and should be rendered with large view distances. Other modes of NPR have been created in past for terrains. Pen and Ink approaches [12, 29] exist which mostly focus on silhouette of the terrain. These are, however, different than painterly rendering process which is the focus here.

Chapter 3

Real-time Terrain Rendering

Terrain data is basically a 2D grid of heightmaps between which triangles can be rendered to give it a rigid water-tight look. However, the number of heights can be very large and drawing all the triangles all the time is not practical on a real-time system. We apply different methods to achieve a real-time performance. In Section 3.1, we see how terrains are divided into tiles and how tiles are kept in the memory. Section 3.2 shows how tiled structure helps in eliminating regions of terrain which are not in the camera's view. Next section, Section 3.3 shows the method of further improving the performance by exploiting loss of detail of farther regions of terrain in view. Regular tiles become very small at the far extremities of the viewing frustum. We take care of this problem by using low levels of detail for such tiles in view. However usage of level of detail brings another problem of terrain popping which happens due to change in level of detail when the camera is moving. We calculate blending factors on a per tile basis and is used to smoothly change the level of detail. In section 3.5 we see another problem of cracks introduced by the LOD system, which is due to difference in LOD of surrounding tiles. We address this by stitching surrounding tiles. Section 3.4 finally describes the rendering method which works upon the described methods. We describe the various steps involved in first creating our terrain representation and then rendering it.

3.1 Data Organization

Terrain data consists of a height value for every point x, y on a rectangular grid. We divide it into **tiles** of equal size for rendering. By equal we mean they cover the same rectangular area on the heightmap. To handle levels of detail, we arrange the data in a specific way. For a tile with size $2^n \times 2^n$ height values, we store m number of LODs, $m \leq n$, m is a user defined number based upon characteristics and size of the terrain. We also keep the distance between adjacent heights in x, y as s_x, s_y Fig 3.1. For an LOD l we have $2^{n-l+1} + 1 \times 2^{n-l+1} + 1$ $(l > 0)$ number of height values and $2^n + 1 \times 2^n + 1$ for $l = 0$. Note the extra heights at the end corners of the tiles, they are the height values at the starting corners of the next tile; kept as they help in stitching (see Section 3.5). This means $l = 0$ holds highest detail and $l = m$ holds lowest detail as illustrated in Fig. 3.1. For $l > 0$ we keep original height values

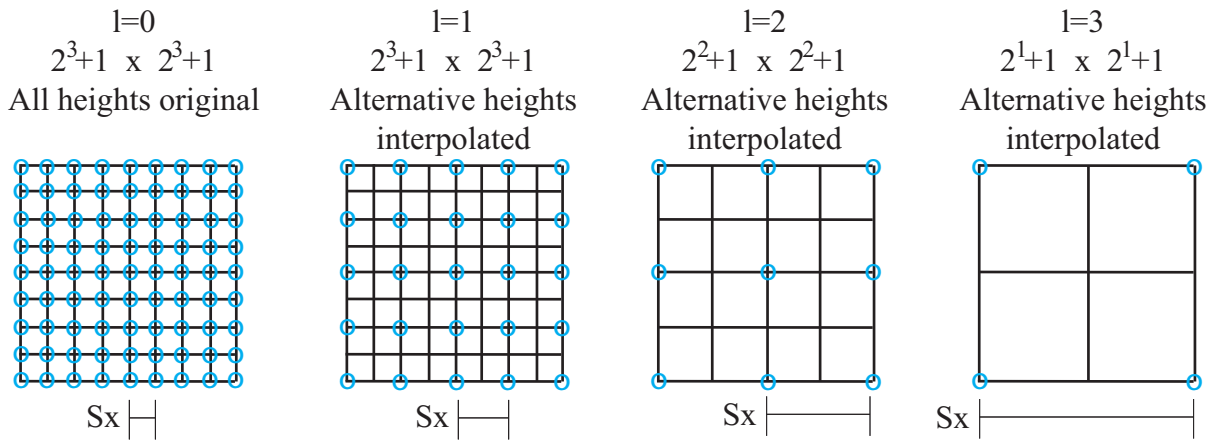

Figure 3.1 Data organization: An e.g. with $n = 3$ and $m = 3$, blue circled height values are original, rest are interpolated. Note that, they occupy the same area on ground

h at $(2i, 2j)$ locations, $0 \leq i, j \leq 2^{n-l}$. We replace the height values at $(2i, 2j + 1)$ locations with $avg(h_{2i,2j}, h_{2i,2j+2})$, at $(2i + 1, 2j)$ locations with $avg(h_{2i,2j}, h_{2i+2,2j})$, at $(2i + 1, 2j + 1)$ locations with $avg(h_{2i,2j}, h_{2i+2,2j+2})$; where i, j vary as bounded. This is done so that while rendering when LOD l with alternate height values dropped, we don't see any change in the structure.

3.2 View Frustum Culling

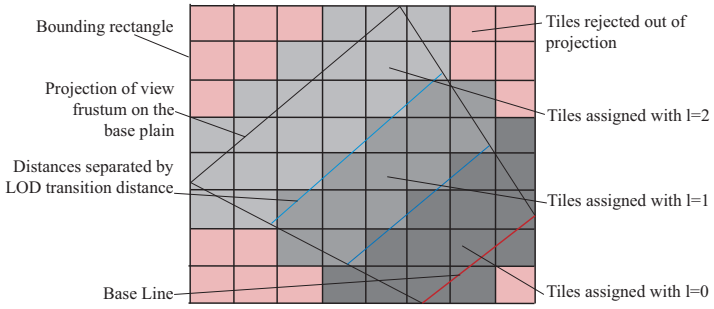

Figure 3.2 View frustum culling and LOD assignment

In each frame, we query the graphics API for view frustum equations and calculate the projection P of the frustum (generally a trapezoid) on the base plain. This base plain is $z = a_h$, a_h is the approximated average height of the terrain in view of previous frame. This is because we haven't accessed the terrain data yet and thus will be using the data from previous frame assuming that the view hasn't changed much. We then calculate orthogonal bounding rectangle of P. We can directly map the coordinates of the bounding rectangle to tile indices. Using these tile indices, we find other tiles that are inside P (Fig.3.2). We keep the indices that return positive in a tile buffer B_t for use in rendering. We do not need to do 3D view frustum culling as terrains are injective functions on x, y, and thus can be reduced to 2D in turn to reduce number of required calculations.

3.3 LOD and Blending Factor Calculation

Using the camera parameters we calculate a base line, that is perpendicular to the view vector and parallel to the ground plane. For each tile in B_t, we calculate the perpendicular distance d of its mid point from this line (Fig.3.2). This distance d is used to calculate LOD l as $\lfloor d/l_t \rfloor$ where l_t is the LOD transition distance. We choose this distance d instead of the direct distance of the tile from the camera because if the field of view of the camera is high, we shall end up rendering tiles at the corner of screen that are actually close to camera in screen space but far in object space in very low level of detail. The value $frac(d/l_t)$ is the blending factor α. α is used for smooth level of detail changes of tiles as explained in Section 3.4. We save l and α in B_t along with the tile indices.

3.4 Rendering

With all data in place, the tiles can be rendered from B_t. For all tile indices in B_t, we load the level l and $l+1$ of that tile. The index is clamped to m to avoid memory exceptions. The distance between adjacent heights for l can be calculated as $(s_{x_l}, s_{y_l}) = (s_x, s_y)2^l$ Fig 3.1. We calculate the heights h for $l > 0$ as

$$h = h_{(2i,2j)_l}(1-\alpha) + h_{(i,j)_{(l+1)}}\alpha$$

$l = 0$ is a special case: $h = h_{(i,j)_0}(1-\alpha) + h_{(i,j)_1}\alpha$, i, j vary as bounded. We can now see that when α is 0, $h = h_{(2i,2j)_l}$, and when α is 1, $h = h_{(i,j)_{(l+1)}}$. Thus this blending factor is able to smoothly change between the two height values of 2 different LODs of the same tile as we move the camera. On the fly, we also calculate the average of the heights at the mid point of these tiles, a_h, which will be used in the next frame for view frustum culling (See Section 3.2).

3.5 Tile Stitching

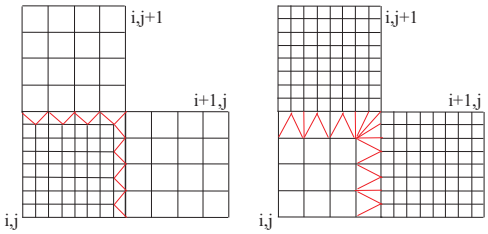

Figure 3.3 Tile Stitching: tile i, j is stitched only to $i, j+1$ and $i+1, j$

Since every tile is getting assigned l and α independently, we find un-tessellated areas near the corner of each of the tiles. We assume that a tile on the ground with LOD l can have a nearby tile whose LOD can be only $l-1$ or $l+1$. This makes tile stitching easy and smooth blending of LODs works perfectly. Our assumption remains true iff l_t is always more than the maximum distance a tile can extend on the

ground, i.e., the tile is never able to skip an LOD in between. So for a tile index t_i, t_j in B_t, we get the l and α of $t_{(i+1)}, t_j$ and $t_i, t_{(j+1)}$, and use them for the corner heights of t_i, t_j Fig 3.3. Note that we are not looking at $(i-1, j), (i, j-1)$ indices of tiles since those corners are already stitched by earlier tiles.

3.6 Conclusions

In this chapter we presented a fundamental terrain rendering system which introduces the categories of problems involved. Using the representation of tiled structure of terrain, we were able to do a quick view frustum culling and level of detail management. With this system we showed how important role a representation plays in designing a good rendering scheme. This chapter shows a very simple terrain renderer, is limited to theory and does not address the issue of handling very large terrains. In the following chapters, we will address practical problems and discuss adaptivity of the techniques to the use of current graphics hardware.

Chapter 4

Terrain Rendering and Manipulation using GPUs

We present a scheme to render terrains, deform them, edit them, and perform physics involving them at real-time rates. We use a representation that combines the fixed-size structure of geometry clipmaps and the regularity of tiled blocks. The terrain is cached on the GPU using fixed-size rectangular blocks. The resolution of the blocks depends on the view and changes with height of the camera. A blocked, tiled, height-map representation resides at the GPU cache at all times for fast rendering and real time modification. The cache is kept updated in extent and resolution by sending data when needed.

The main contributions of this paper are: (a) A terrain rendering system that achieves a rendering speed of 100 frames per second on arbitrarily large terrains without the CPU, the GPU, or the bandwidth between them being the bottleneck. CPU load is less than 10% while rendering at 100 fps (b) A way to interactively modify and interact with the terrain simultaneously with rendering, performed entirely in the GPU at real-time rates. This enables terrain deformations, interactive editing and the computation of simple physics of external objects interacting with the terrain. The following innovations make the above possible. (i) A terrain representation that uses fixed-size blocks of grids and GPU caching that enables fast rendering and correct editing and manipulation. (ii) A scheme of sending light geometry templates from the CPU to the GPU, which are expanded into the actual geometry. This keeps the CPU free to do other tasks while the GPU performs the bulk of the rendering work. (iii) A two-level culling scheme with the CPU culling in units of large tiles and the GPU culling in units of smaller tilelets for high rendering performance. The rendering rate doubles with this. (iv) Clever interleaving of data transfer from the CPU to the GPU to keep the cache updated correctly without affecting the rendering rates. This guarantees 100 fps rendering of arbitrarily large terrains. (v) Fast, interactive and procedural manipulation and editing of GPU-resident terrains using the fragment shader. The highly parallel GPU resources are employed profitably to do this, improving the system performance.

We demonstrate the performance of our system using an Nvidia 8800GTX GPU. We can fix the framerate at 100 while handling a $1M \times 1M$ terrain which uses 2 TB for the heights[1]. Our system renders upto 350 million triangles per second on parts of a flight path and achieves an average rate of

[1]The large terrain is a periodic extension of the $16K \times 16K$ Puget Sound terrain. The terrain system is unaware of the replication. The CPU module that loads the terrain is aware of the fact and returns pointers to existing data when going beyond

160 million triangles per second. The frame rate of 100 can be maintained even while half the terrain is deforming or is being edited or when 256K balls are bouncing on it. We exploit the advanced SM4.0 features of the GPU to achieve the high performance [43].

The terrain representation and rendering are explained in Sections 6.2 and 4.2 respectively. The caching system is explained in Section 4.3. Terrain manipulation schemes are presented in Section 4.4. Section 6.5 presents experimental results. Some concluding remarks are given in Section 4.6.

4.1 Terrain Representation

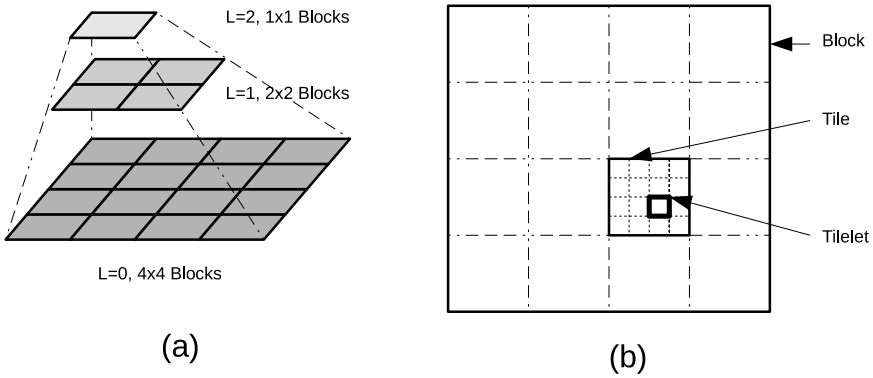

Figure 4.1 (a) An terrain with highest resolution stored in 4×4 blocks, next in 2×2 blocks and so on, using fixed size blocks. (b) A block with 4×4 tiles each with 4×4 tilelets

We represent the terrain as a regular 2D grid of heights with a fixed post distance in X and Y directions. Fixed-size *blocks* are used as the base units of storage and transfer from the CPU to the GPU. A block consists of *tiles*, which are the basic rendering units. Tiles extend in the ground XY plane and take part in view frustum culling. Tiles are further divided into smaller *tilelets* (Figure 4.1(b)).

CPU Representation: The terrain is stored in main memory and sent to GPU as blocks. We use blocks of size 1024×1024 as larger blocks are more efficient for transfer. The CPU also stores all lower resolutions of the terrain as blocks of the same size to facilitate quick transfer of arbitrary levels of detail of the terrains to the GPU (Figure 4.1(a)) at the cost of a maximum of $1/3$ more memory. Thus, a terrain of 32×32 blocks at the highest resolution needs 16×16 blocks in the next lower resolution. View frustum culling takes place at finer resolution in terms of tiles. Tile size should balance the culling and rendering loads. We use larger tiles in our system, currently 256×256, since geometry is discarded also at the GPU as explained later.

Different parts of the terrains need to be rendered at different levels of detail or resolutions. The *level-of-detail (LoD)* at which a tile is rendered depends on two factors: the elevation of the viewpoint and the distance along the ground from the viewpoint. The rendering resolution reduces as the elevation E

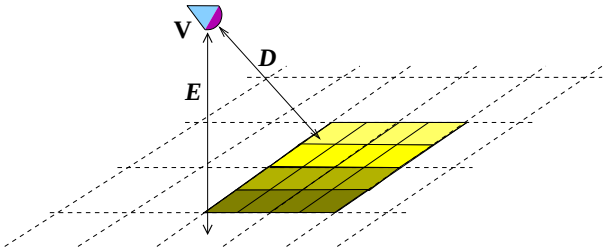

Figure 4.2 The GPU Cache (shown shaded) is a contiguous section of the terrain residing in the GPU as a regular grid at a resolution determined by the elevation E of the viewpoint V. The rendering resolution also depends on the distance D.

or the distance D increases (Figure 4.2). Resolution is changed in discrete steps by doubling or halving the post-distance. *Merge level* denotes the level-of-detail of the terrain determined by the elevation and *distance level* denotes the same due to the ground distance to the viewpoint. Both take discrete integral values. Level 0 represents the terrain at the highest detail; level $i + 1$ represents the terrain at half the resolution of level i. The LoD is expressed as the number of level shifts from the highest resolution available. The *rendering LoD* of a tile is the sum of merge and distance levels.

The transition distance in elevation for shifting the merge level depends on the image-size to which a triangle projects. We shift the merge level when the post distance maps to about 3 pixels (see [38]). For a 90 degree field of view and a 1024×768 window, the transition distance comes out to be $1024/(3 * 2 * \tan \phi/2)$ – about 170 times the post-distance – when the viewpoint is close to the ground. The transition distance doubles with each reduction in resolution as the post-distance doubles. This scheme can handle terrains of very high resolutions. Since the transition distance depends on the post-distance, the higher resolutions will be used only when the viewpoint is closer to the ground. We use the same distance for ground transition distance for near uniform behaviour due to camera changing elevation or distance. Farther tiles render with low detail or higher LoD number.

Resolution reduction is achieved easily on our representation by dropping alternate rows and columns. Thus, an LoD level l has a post distance that is 2^l times the post distance at level 0. Thus a higher resolution block contains all lower resolution ones, which can be generated by sub-sampling. We choose sub-sampling instead of filtering for creating low resolutions because it preserves height values whereas filtering changes the heights in lower resolutions. Sub-sampling is also fast but produces no artifacts when combined with our blending scheme explained in the next section.

GPU Representation: The *GPU Cache* holds a contiguous 2D grid of blocks at the merge-level resolution around the *point of reference* determined by the camera location. The resolution depends on the view elevation E (Figure 4.2) and is the highest resolution needed for rendering from that elevation. It also contains all lower resolutions as explained before. Thus, all data for rendering the terrain is entirely in the GPU memory. The appropriate LoD for each tile is generated on the fly by sub-sampling. Tiles are further divided into 2×2 *tilelets* by the GPU and used for the finer *two-level of culling* explained later. The GPU Cache is updated when the merge level changes due to elevation or the region changes

due to change in camera location. It is to be noted that the extent on the ground of the blocks and tiles change with the merge level as they have fixed sizes.

Implementation Details: The GPU Cache is stored as an *array texture*, introduced in SM4.0. The cache can be attached to a single texture unit and a height can be accessed on the fly by the GPU without periodic texture binding by the CPU. Heights are accessed using three coordinates: l to select the block (also called a layer) and x, y to fetch the post from that layer. Each layer of the array texture can be updated randomly. We use a separate *pointer-texture* to store the layer IDs (Figure 4.3). The pointer-texture is a 2D array of layer IDs and presents the GPU Cache as a contiguous 2D array of blocks. The l coordinate is fetched using the 2D indices of the block. The pointer-texture is updated with new layer numbers when the GPU cache is updated. The unified architecture of SM4.0 provides fast access to the texture for all shader units.

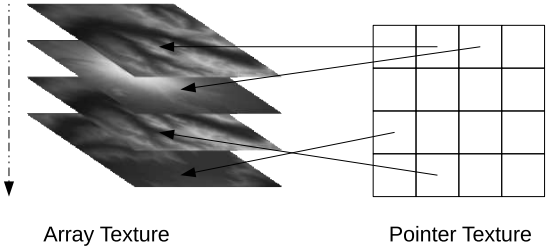

Figure 4.3 An array texture with 16 layers and the GPU cache as a 4×4 pointer texture that stores the layer IDs.

The blocks and tiles have fixed memory sizes and variable extents on the ground. A block at merge-level $l + 1$ spans four times the area of the same at level l. The merge and distance levels provide a unified LoD scheme with nearly constant triangle count on the screen for all elevations of the camera. The amount of data to be rendered also is nearly constant at all elevations due to the shift in resolution. This plays an important role in achieving real-time performance on large terrains. Our system also supports mapping of real texture images to the terrain. These textures are kept in a parallel cache on the video memory with a matching block of texture for each block in the GPU cache. The resolution of the texture blocks is kept updated for seamless performance. Each block is mipmapped independently to avoid aliasing for far-off tiles. Hardware mipmapping may not produce the desired results as the screen space measure used by it may not match the resolution reduction used for the terrains. We currently store the complete texture images in multiple resolutions on the video memory. The texture cache stores the coordinates of windows to it.

4.2 Terrain Rendering

The GPU performs most of the rendering under CPU's coordination. The CPU culls every tile in the GPU cache to the view frustum. It them sends the geometry template, consisting of a vertex buffer

object (VBO) of points, for each tile to the GPU. This keeps the CPU load the communications to the GPU very low. The GPU discards tilelets of the geometry that lie outside frustum and expands the rest into the triangles.

4.2.1 Stage 1: CPU

The 2D grid scheme makes it easy to compute the extents of tiles, blocks, and tilelets using simple calculations. Each tile has an index in the grid of tiles. The CPU eliminates tiles outside the view frustum and computes the LoD level for the rest of the tiles. It then sends the corresponding geometry templates to the GPU.

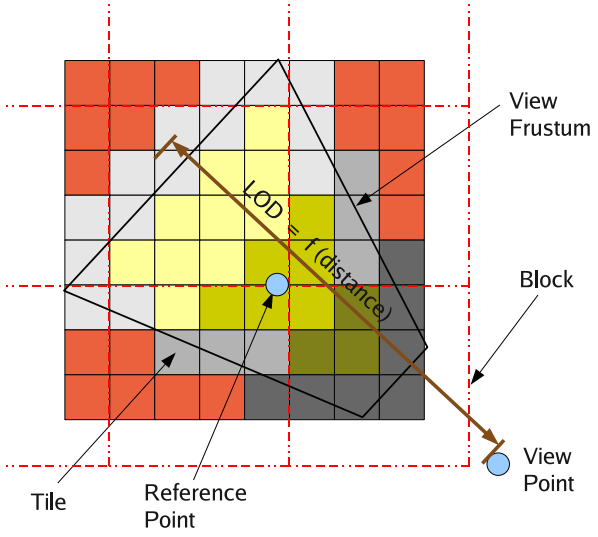

Figure 4.4 Tiles outside view frustum (marked red) are discarded by the CPU. Intersecting tiles (gray) will go through a second level of culling by the GPU. Interior tiles (yellow) are rendered directly. LoDs of tiles to be rendered is a function of distance from the viewpoint.

View Frustum Culling: The orthogonal footprint of the view frustum and its bounding box are estimated in the grid of tiles first, as shown in Figure 4.4. The bounding sphere of each tile is tested against the six planes of the view frustum and those lying outside are discarded. Tiles that intersect a frustum wall are tagged specially boundary tags as their tilelets will undergo a second level of culling in the GPU.

Level of Detail: The GPU cache holds the terrain at the current merge-level. The CPU uses the distance of each tile from the camera to compute the distance-level. The distance-level of each tile thus denotes the drop in resolution from the data stored in the GPU cache. Farther tiles will get a higher LoD number or lower detail. Thus, proximate tiles will be rendered with high detail and distant ones with low detail (Figure 4.4). The detail factor d_l is computed as $\log_{1.5}(1 + \frac{d}{t})$, where d is the distance E of the mid point of the tile from camera (Figure 4.2) and t is the current diagonal length of the tile. The term d/t will give linear LoD bands and the logarithm will ensure exponential thickness for equi-detail

bands. This results in near-uniform distribution of tile-detail on the screen. The base of the logarithm affects the width of the LoD bands. A value of 1.5 gives acceptable triangle count and quality in our experience and ensures a minimum LoD band thickness of one tile. Thus, adjacent tiles will not differ by more than one level which is necessary for seamless stitching as explained later. The integer part $\lfloor d_l \rfloor$ of the detail factor is used as the distance-level l_d to shift the resolution and the fractional part is used as the *morphing factor* α for the entire tile. Morphing of different LoDs is necessary to avoid popping artifacts as explained later.

Rendering: The CPU sends each tile to the GPU along with l_d, the morphing factor, and the boundary flag. To reduce CPU load, a *geometry template* of a VBO of point primitives is sent for each tile. Each point of the template represents a tilelet to be rendered. We currently use 2×2 tilelets. A 128×128 VBO is used for the full 256×256 tile. Smaller VBOs are used if the distance-level is greater than 0. The same template can be used for all tiles at a particular distance-level. This process is explained in detail in Section 4.2.2.

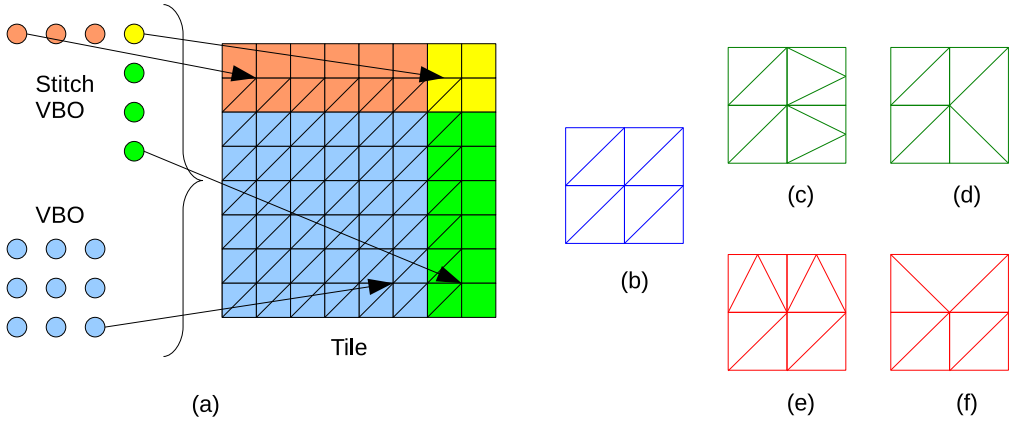

Figure 4.5 (a) The CPU renders each tile (here of size 9×9) as points using two geometry templates, one for the interior (shown in blue) and the other for the boundary (red/yellow/green) of the tile. (b) Tilelet used in the interior of the tiles. In the eastern border (green), tilelet (c) is used when the neighbor has a higher LoD and (d) is used if lower. In the northern border (red), tilelet (e) is used when neighbor has a higher LoD and (f) is used if lower. Yellow region gets handled automatically.

Adjacent tiles can have different resolutions which causes visual inconsistencies at the joints. We use a *stitching* process for border rows and columns to avoid this. Each tile stitches with its northern and eastern neighbors as explained later. Stitching requires an extra row and column of indices of the neighbor to be available to each tile. Thus, the actual tile size used is 257×257 with its last row and column being the first row and columns of the adjacent tiles. CPU sends separate *stitch templates* to effect correct stitching (Figure 4.5(a)).

4.2.2 Stage 2: GPU

The GPU receives a point for tilelet of a tile being rendered along with its LoD and morphing factors. First, the tilelets outside the frustum are discarded, resulting in a fine culling in terms of 2×2 sections of the terrain. Geometry is generated for the surviving tilelets by accessing the terrain from the GPU cache.

Tilelet Generation and Culling: The GPU receives the index, the LoD number, the morphing factor, and geometry template for each tile (Figure 4.5(a)). The index is mapped to the block index of the pointer texture and the layer and tile ID of the GPU Cache. The coordinates of the incoming point primitive of the template represents the top-left corner of the corresponding tilelet and the LoD can be used to compute the other corners. The heights of the corners are fetched from the GPU cache. These four points are tested against the view frustum by the vertex shader. The tilelet is tagged as outside if all four points are outside before sending down the pipeline. In practice, conservative testing is performed to counter possible error introduced by the quadrilateral approximation of the tilelet. This process of second level culling is performed only on the tiles that intersect the view frustum walls as tagged by the CPU. Other points are passed down the pipeline without testing.

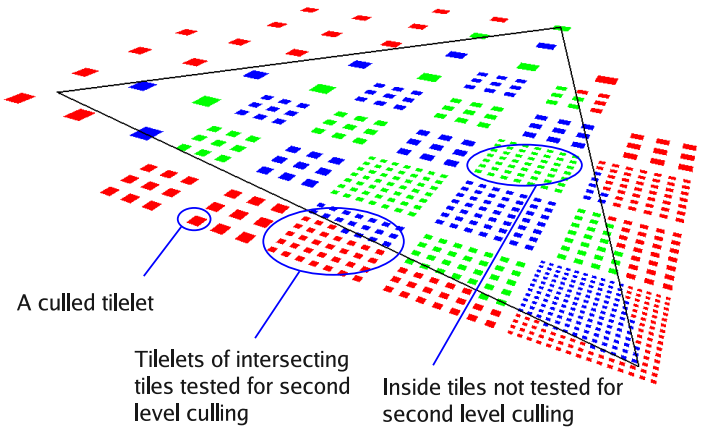

Figure 4.6 Picture of tilelets after VFC. Farther tiles need fewer tilelets. The red tilelets are discarded by the second level culling on the GPU.

The geometry shader of the GPUs can discard primitives from the pipeline or add primitives to it. The tilelets that are tagged by the vertex shader are discarded. This second level culling accomplishes accurate culling with no load to the CPU. Figure 4.7 shows that the frame rate is doubled if second level culling is used on a typical rendering. We experimented with different tilelet sizes. The smallest tilelet with stitching capability has a size of 3×3. This performs the best due to the deterioration in performance of the geometry shader as the amount of data it outputs increases [17]. The geometry shader generates triangles for the remaining tilelets. The coordinates of the top left point and the LoD number are used to access the 3×3 grid points. The post distance at level l is 2^l times the post distance at level 0. Triangles of the tilelet, as shown in Figure 4.5(b-f), are sent down the pipeline.

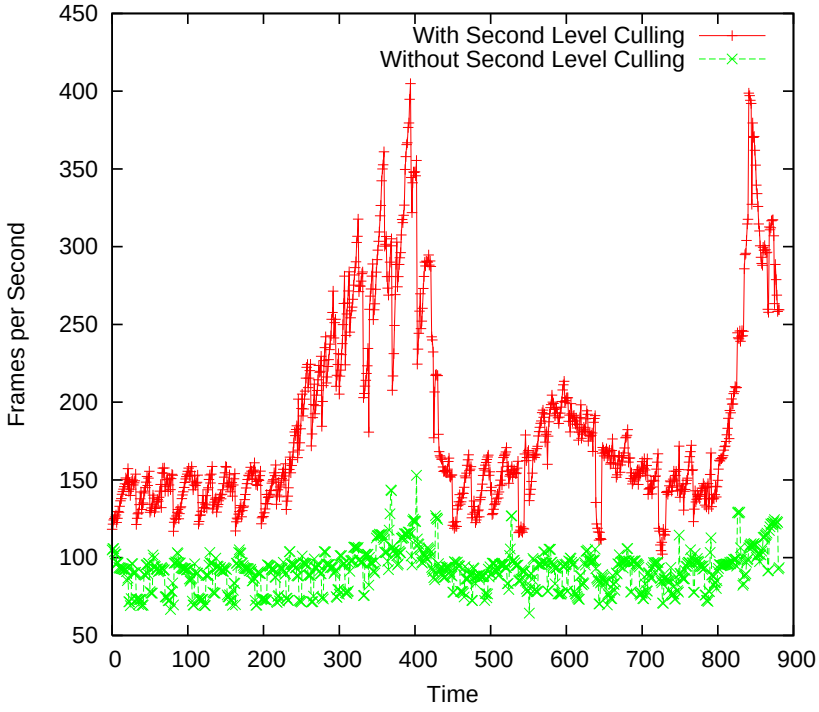

Figure 4.7 Framerates for a typical flight over the terrain with (red) and without (green) the second level of culling.

Figure 4.7 shows the performance of the system over a typical flight over the terrain with and without the second level culling. The second level culling improves the system performance by a factor of 2 overall. The camera is looking approximately down between frames 250 to 450. The tiles nearer to the camera contain more tilelets as seen in Figure 4.6. The second level culling is most effective on them as a result.

4.2.3 Tile Stitching and Blending

Tiles can be rendered independently if the northern row and eastern columns are included. However, stitching is necessary when the northern or eastern neighbor has a different LoD. Gaps in the terrain may be visible otherwise. Zero area triangles have been used to alleviate this problem in the literature [38], but is an inelegant and incorrect solution. We use a special L-shaped geometry template with indices of only the northern and eastern borders of a generic tile for stitching (Figure 4.5(a)). The CPU sends these templates with tile indices, LoD number, morphing factor as well as the LoD number and morphing factor of the neighbouring tiles.

The stitching templates use the same vertex shader. Their geometry shader selects one of the tilelets in the border areas (Figures 4.5(b-f)) based on the LoD numbers. Tilelets of the eastern border can be rendered with equal LoD to the parent tile (Figure 4.5(b)), lower LoD than the parent tile (Figure

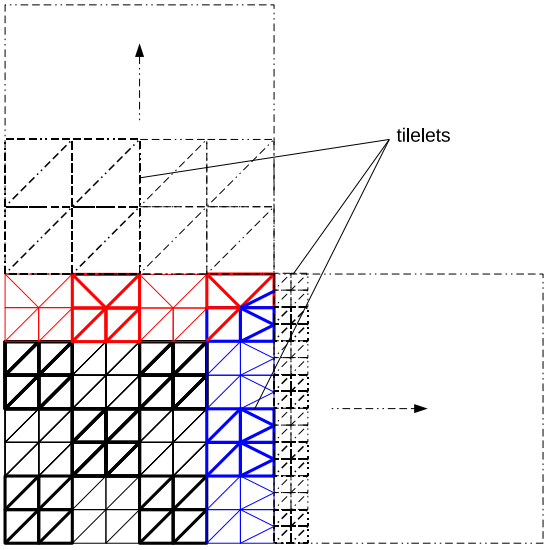

Figure 4.8 A tile of size 9×9 with a northern neighbor of lower resolution and an eastern one of higher resolution. The tilelets of Figure 4.5 are used for correct stitching.

4.5(d)), or higher LoD than the parent tile (Figure 4.5(c)). The same goes for the tilelets along northern border (Figure 4.5(b, f, e)). The geometry shader can recognize and render these tessellation styles using the available information. The north-eastern tilelet which is adjacent to both neighbours gets handled automatically. An example tessellation of a tile is shown in Figure 4.8. Our stitching scheme maintains coherence at the borders of tiles with no abrupt changes or gaps. It also requires less number of triangles compared to schemes like zero area triangles [38] and introduce no extraneous geometry.

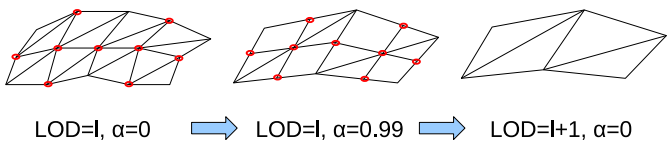

Figure 4.9 A tile at LoD = l (left) blends its alternate heights (shown in red) with its lower LoD (middle) using α. When the tile shifts its LoD, the change is not noticeable. This process is valid in reverse as well.

The morphing factor is used to smoothen the change in LoD to avoiding popping of geometry. The morphing factor α has a range $[0, 1)$ and is used to interpolate between the heights of the current LoD level and of one lower level. Thus, the final height used for rendering is $h = \alpha h_l + (1 - \alpha)h_{l+1}$ where l is the LoD of the tile (Figure 4.9). As a tile of LoD l moves closer to farther edge of its LoD band (due to camera moving back), its α will drop to 0. When it crosses into the next band, the LoD will shift to $l + 1$ and the alpha will become 1. When a tile moves from far to near, α changes from 0 to

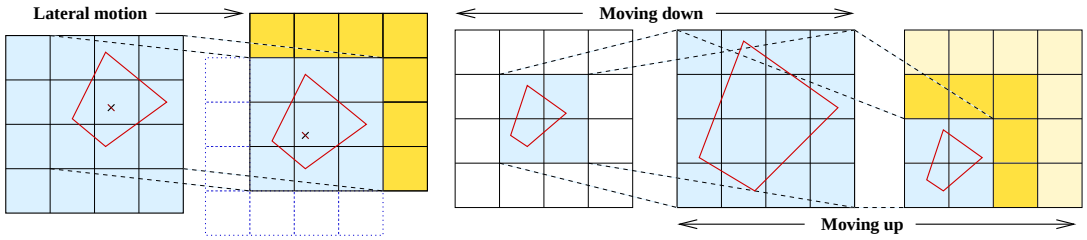

Figure 4.10 Later motion and pan/tilt (left) involve discarding an L-shaped region and bringing in new blocks (yellow) from the CPU. When the viewpoint comes down, the merge level decreases (middle). The extents of GPU cache are halved and data at a higher resolution is brought in from the CPU. When the viewpoint goes up, the extents of the cache are doubled and the existing data is compressed into one quadrant. New data is brought in from the CPU with nearer blocks getting higher priority in a staggered manner (right). The data transfer is scheduled in a staggered manner not to affect rendering speed.

1 smoothly and increases the resolution of the tile. For the corner heights, the morphing factor for the adjacent tile is used, as the neighbouring tile will change its LOD independently. This is the difference when rendering the stitching triangles. This blending scheme ensures a continuous transition between different resolutions.

4.3 Caching

The GPU Cache contains $N \times N$ blocks at the merge-level resolution, which is the maximum resolution of the terrain needed at the elevation of the camera. The size N depends on the maximum visibility required at the highest resolution. The visibility doubles as the merge-level increases. We use $N = 8$ for most of our experiments, needing storage for 64 blocks on the GPU. The cache is updated periodically to hold all the data needed for rendering when viewpoint changes. One option is to keep the GPU cache symmetric with respect to the viewpoint as is done by Losasso and Hoppe [38]. This ensures that all data to look around from a particular point is present in the GPU cache. We, instead, try to keep the GPU cache symmetric with respect to a **reference point**, which is the centre of the orthographic projection of the view frustum onto the ground (Figure 4.10).

4.3.1 Lateral Motion of Viewpoint

Lateral motion, pan, and tilt at a constant elevation bring in new data to the GPU cache at the same resolution. We use the position of the reference point in the cached terrain to trigger the data transfer. If the reference point goes outside the central $n \times n$ block of the GPU cache – where $n = N/2$, – the cache is re-centered by bringing another row or column of blocks at the current merge level, (Figure 4.10) discarding blocks on the other side of the cache. We load the new blocks by overwriting the discardable blocks. The data from the CPU is loaded to selected layers of the array texture and the pointer texture is updated to rearrange the layer IDs on the GPU. For an 8×8 GPU Cache with each block taking 2MB

of memory (1024×1024, 16-bit height values), a lateral motion cache update needs 16MB of data to be uploaded to the GPU. The data transfer time is controlled using a *job-queuing* scheme explained later.

4.3.2 Vertical Motion

When the camera goes up, the extent of the visible terrain increases and the resolution decreases. Similarly when the camera comes down, the terrain extent decreases and the detail increases. For this, we change the merge level or the base resolution of the GPU Cache. This process keeps the memory footprint constant without compromising the requirements for rendering.

Ascending Motion: The merge level increases and the resolution halves when the viewpoint moves up. A quarter of the GPU cache can be filled by sub-sampling and merging the current contents of the cache (Figure 4.10). New data has to be brought to the remaining space. The merging is performed on the GPU using a separate fragment shader pass that sub-samples and copies heights from 2×2 blocks into a single unused block. This is achieved by the binding the target and source blocks as frame buffer objects or textures and drawing a block-sized quad. This process needs one extra block of storage on the GPU. The array texture thus has an extra layer which is kept in an *unused-layer queue*, outside of pointer-texture. When merging, an unused layer is dequeued and used as the target. The original four blocks are queued as unused after the merge. At the end of the merging process 75% of the blocks will be free (Figure 4.10). New data is brought to them from the CPU in the proximity order from the reference point and stored in unused layers. The data transfer is triggered before the new area is needed and can complete over a few frames. After the process completes, the GPU Cache will have $N \times N$ blocks and one unused layer. For an 8×8 GPU Cache, we merge 64 blocks into 16 blocks in the GPU using 16 merge operations. After that, 48 blocks or 96MB are uploaded from the CPU to the GPU.

If the original terrain is one million square, we can get $\log_2 1024 = 10$ global LoDs. We reduce the resolution in factors of 2 until the entire terrain fits into the GPU Cache. Thus the number of merge-levels depend on the cache size and the total size of the terrain. We use a GPU Cache size of 8×8 of $1K \times 1K$ sized blocks for the $16K \times 16K$ Puget Sound data. It contains 14 LoD levels and only one ($= \log_2 16384 - \log_2 8192$) merge-level before the GPU cache is filled. A $1M \times 1M$ data with the same cache can use 7 ($= \log_2 1M - \log_2 8192$) merge levels. The data transfer time is controlled using a *job-queuing* scheme explained later.

Descending Motion: When the resolution increases due to a reduction in elevation, the blocks of the cache are replaced by higher resolution blocks, and the total extent of the terrain reduces. This is data intensive as the entire GPU Cache needs to be replaced.

A quarter of the GPU Cache that will remain in the view are first identified. The physical area of each block is to be replaced by four high resolution blocks. The remaining layers are added to the unused-layer queue (Figure 4.10). Each block is replaced with its 4 higher resolution descendents in order. As soon as four blocks are loaded using the unused layers, the pointer texture is updated and the low resolution block is enqueued as unused. When the process is finished, we will have $N \times N$ layers in the GPU cache and one unused layer. The higher resolution is not needed for rendering immediately

since distance LoD increases smoothly. The increase in resolution is also anticipated ahead of time to avoid visible update changes. For a 8×8 blocks, we have to bring the 128 MB into the GPU memory to increase the merge level resolution. The data transfer time is controlled using a *job-queuing* scheme explained next.

4.3.3 Job Queuing Scheme

Our primary goal is to maintain a steady frame-rate. We treat the data transfer and merge operations as "jobs" and queue them to be executed when the rendering process has time. Running all the operations at the same frame can freeze the rendering at times and affect the quality of visualization. We execute as many jobs as possible to keep the total frame time T_t, safely within the fps constraints. The total frame time is, $T_t = T_r + T_u$ where T_r is time for rendering and T_u is the total time taken by the jobs in the cache updating process. For 100 fps rendering, T_t is 10 ms and we are left with $T_u \leq 10 - T_r$ ms for updating. We steal cycles for the update process when $T_r < 10$ms without affecting the fps.

The transfer of a block of 2MB from the main memory to the GPU takes 2 ms on the current GPUs. A merge operation takes less than 0.5 ms. The number of jobs to be performed is calculated as $n = (10 - T_r)/K$ where K is a constant denoting the worst case time for the job. For example if $T_r = 5$ ms, and $K = 2$ ms for a layer update, then $n = 2$ jobs can be performed per frame. If $n < 1$, we do half jobs, by uploading half of a block. Over some number of frames, all operations are completed. This adaptive job-queuing ensures a frame-rate of 100 in practice without any hiccups.

For the cache to be completely updated before the next updating is necessitated, the user speed will be limited. For lateral motion, the worst case scenario is if all the blocks are updated by half jobs needing 8 jobs or 16 frames or 0.16 seconds. If a block spans X kms at the current merge level, then the speed limit on the camera is $X/0.16)$ kms/seconds. In case of ascending motion, a worst case scenario occurs when data uploading is done using half jobs, needing 96 frames or 1 second to complete. The speed will be limited to X kms/second. In case of downward motion, if all 64 blocks are updated using half jobs, we need 128 frames, restricting the user speed to $X/1.28$ kms/second. As we go up, the blocks span double the distance on ground and thus the speed doubles for the next merge level. It is clear that the vertical motion determines the speed limit. However, horizontal motion is most common during terrain visualization and hence the restrictions are not limiting. For a post-distance of 10m for the Puget Sound data, 8×8 blocks provide a visibility of up to 80 kms when the viewpoint is close to the ground. With a block's span of 10 kms, the horizontal speed limit comes to $10/0.16 = 62.5$ kms/second and $10/1 = 10$ kms/seconds in case of vertical motion. Both are quite acceptable.

When the merge-level changes, the GPU cache gets updated completely over a finite time. Until then, artifacts can appear since the cache is mixed with old and new layers. We use *dirty texture* flags to handle this. As soon as the merge-level changes, all the blocks are marked *dirty*. When marked dirty, the renderer uses the lower resolution, as with the old merge-level. As soon as the new layers get updated and older layers are made unused, new blocks are marked clean. This way the rendering remains free of artifacts.

4.4 Terrain Deformation and Manipulation

Terrains are traditionally used only as static geometric entities. The rectangular grid representation makes it easy to manipulate them interactively or procedurally as well as to simulate interactions of other objects with it. The height map structure of terrains make them suitable for topology-preserving deformations. However, deforming terrains is computationally expensive as they are massive entities. Deforming and simultaneous rendering of terrains pose a great challenge.

Interactive and Procedural Manipulation: The GPU representation of the terrain that we use lends itself to interactive and procedural manipulation easily, exploit the computing power and architecture of the modern GPUs. A fragment shader can operate on each height value independently or in relation to its neighborhood. The parameters for the deformation process should be given to the shader. This includes the user inputs like the mouse path for interactive editing and relevant parameters for procedural deformation. A deformation pass is triggered on each block by drawing a block-sized quad after setting up the parameters. Deformation passes are sandwiched between rendering passes. This simulates terrain dynamics in regular frame intervals.

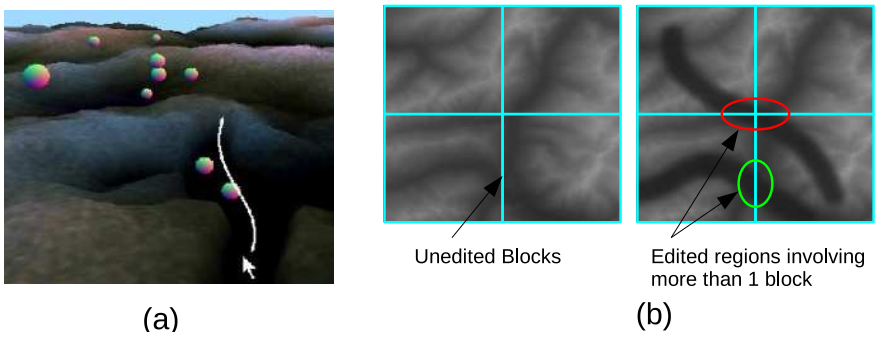

Figure 4.11 (a) The mouse motion over the screen triggers interactive editing of the terrain. (b) A terrain of 2×2 block (left) and the results of editing it (right). Editing can involve multiple blocks at boundaries (shown circled)

Interactive editing of the terrains can also be performed with the user guiding the change in heights (Figure 4.11(a)). The screen point is back projected to world point and transformed to the the terrain coordinates to get a point of impact. Heights are modified based on the distance from the point using user selected radius and intensity of impact. Figure 4.11(a) shows how a channel can be cut on the terrain by dragging the mouse. Multiple blocks may need to be edited, based on the point of impact (Figure 4.11(b)). For procedural dynamism based on time and distance, each deformation pass makes incremental changes to the terrain, between rendering passes. The accompanying video shows a crater formation and a sinusoidal wave passing through the terrain. The heights at each point are computed by the fragment processor using a suitable equation and used subsequently for rendering.

A deformation pass takes about 250 microseconds per block. We see no drop in framerates when only a few blocks are modified in each frame. Figure 4.12 shows the plot of the deformation and rendering times as different numbers of blocks are edited per frame. The modification of the terrain happens entirely on the GPU. We start a simultaneous process on the CPU to effect the same changes on the base terrain. This is similar to write-through of memory caches. The CPU can keep up with the GPU for user-guided editing since the CPU load is low. For procedural deformations, only the last state needs to be created on the CPU. This can be performed as the CPU is lightly loaded.

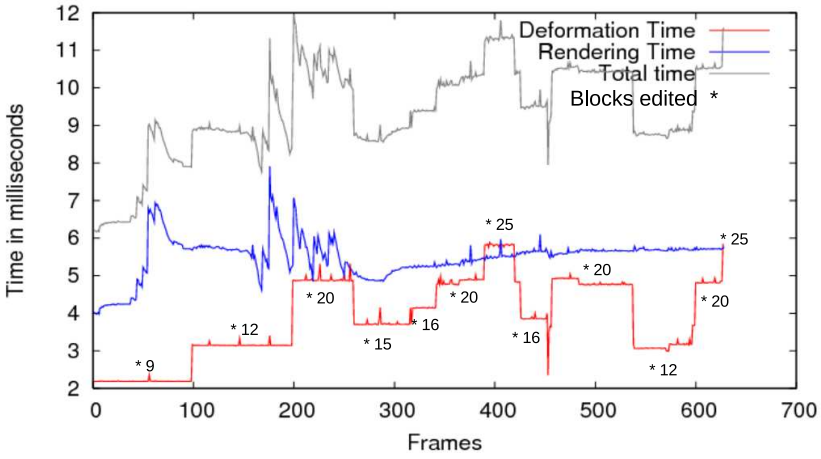

Figure 4.12 Deformation and rendering times for a typical flight over continuously deforming terrain.

The GPU cache is a single array texture. It can be bound as a single FBO and modified in place using a fragment shader. Layered rendering of the current GPUs enables independent editing of multiple blocks in single deformation step. The modified terrain can be rendered immediately as the GPU Cache itself is updated in place.

Real-time Object-Terrain Interaction: Terrains can be used as the base to simulate several interactions with external objects like a bouncing ball. Though the exact physics involved could be quite complex, effective simulation and visualization can be achieved with moderate computation power. We take the example of multiple balls bouncing over the terrain. Balls are modelled as simple point objects are bounce based on the local surface orientation. A gravitation force pulls the balls downward always. If a ball collides with the terrain, local geometry determines its reflected direction and velocity.

The positions and velocities of each ball is stored as two textures in the GPU memory. The positions will be updated by a fragment shader using the velocity and time difference in an update pass that takes place between rendering passes. The fragment shader has access to the textures through an FBO for the update pass. The update pass will also implement the physics such as collision with the terrain. The velocity may change as a result of the physics. The terrain height has to be looked up for a given 3D position of the ball to check for collision. This is done by converting the xy location to the GPU cache block and grid coordinates, looking up the layer ID from the block number using the pointer texture, and accessing the height.

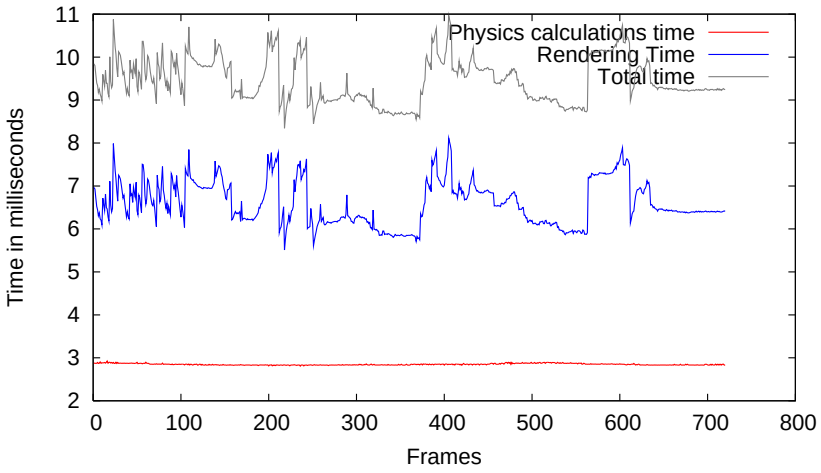

Figure 4.13 Physics computation and rendering times with 256K balls interacting with the terrain. A frame rate of 100 fps is possible.

If balls are present, a rendering pass renders them at their current locations. The vertex shader fetches the positions of a ball, checks for visibility in the frustum and renders it procedurally as a front facing circle. Figure 4.13 shows a flight over the terrain with a quarter of a million balls interacting with it running at 100 fps. The system can achieve 60 fps with 1 million balls[2]. The accompanying video shows visual results from it. Though we use a simple constant velocity model currently, other physics models can also be implemented.

4.5 Results

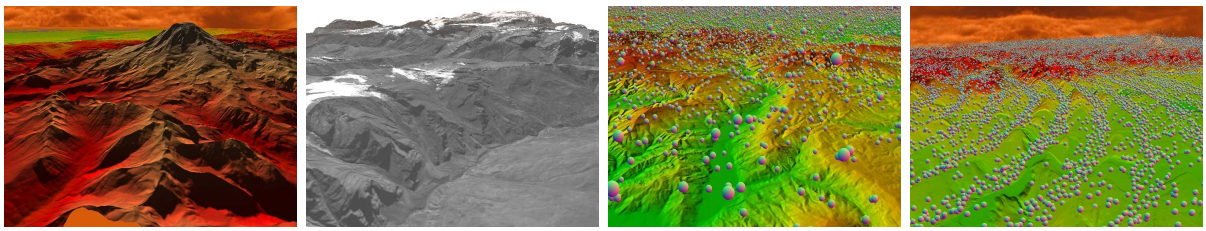

Figure 4.14 A view of Mt Rainier, a terrain with real texture, realtime physics with balls, realtime physics with a deforming terrain.

The overall algorithm is given in Algorithm 1, with parts for the CPU and the shader units on the GPU. We present experimental results using our system. More results are given in the accompanying video. We performed all our experiments on an Nvidia 8800 GTX using OpenGL and GLSL shaders on Linux with a Pentium Core 2 Duo CPU running at 2.4 GHz.

[2]Transform feedback is the recommended mode on SM4.0 GPUs to generate positions and geometry on the fly. Updating the positions using transform feedback is slow and achieves about 17 fps with 4K balls. Fragment shaders are much faster

Algorithm 1 Terrain Rendering

```
Load terrain data as blocks at all LoDs
Create geometry templates for different LoDs as VBOs
for each frame do
  CPU:
    Update GPU Cache if necessary using job queuing
    Discard outside tiles and tag intersecting ones
    Calculate LoD l and blending factor α
    for each tile t do
      Render the geometry template for the interior tiles
      Render stitching template if a border tile
    Vertex Shader:
      If tile is tagged. check tilelets against frustum and tag discards
      Pass on the point down the pipeline otherwise
    Geometry Shader for interior tilelets:
      Discard points tagged discard
      Generate triangles of a 3 × 3 tilelet, with α for morphing.
    Geometry Shader for boundary tilelets:
      Discard points tagged discard
      Generate triangles of a stitching tilelet, with α for morphing.
    Pixel Shader:
      Apply texture, lighting, and fog
    end for
    Initiate deformation pass if editing or deformation is on
    Initiate an update velocity/position pass if physics is on
    Render balls from position texture if physics is on
end for
```

We use the Puget Sound data, consisting of a 16384×16384 grid of 16-bit heights covering a square region of length about 160 km, for most experiments. We also use the 8192×4096 BlueMarble grid with earth texture and an 8192×4096 height data with monochrome satellite image as texture. We simulated a very large terrain by first tiling 4 sets of PugetSound, flipping it along the vertical and horizontal edges. This $32K \times 32K$ terrain occupies 2 GB of space and can be tiled along X and Y directions infinitely. We simulated a $1M \times 1M$ terrain by replicating it 32 times each in X and Y directions. Replication was effected using modulo computation without additional memory. The data-access module on the CPU was the only unit aware of the replication. The terrain system was unaware of it. The accompanying video shows more examples.

Figure 4.15 shows the system performance on a flight over the 1 trillion sample terrain. The camera moves laterally till about frame 2000 with significant tilt. The thin peaks in update time correspond to lateral cache updates. The camera goes up and merge shifts occur near frame 2200. The triangle rate falls when many distance-levels are used as the terrain access doesn't benefit from its caching scheme. The camera starts to come down around frame 3200.

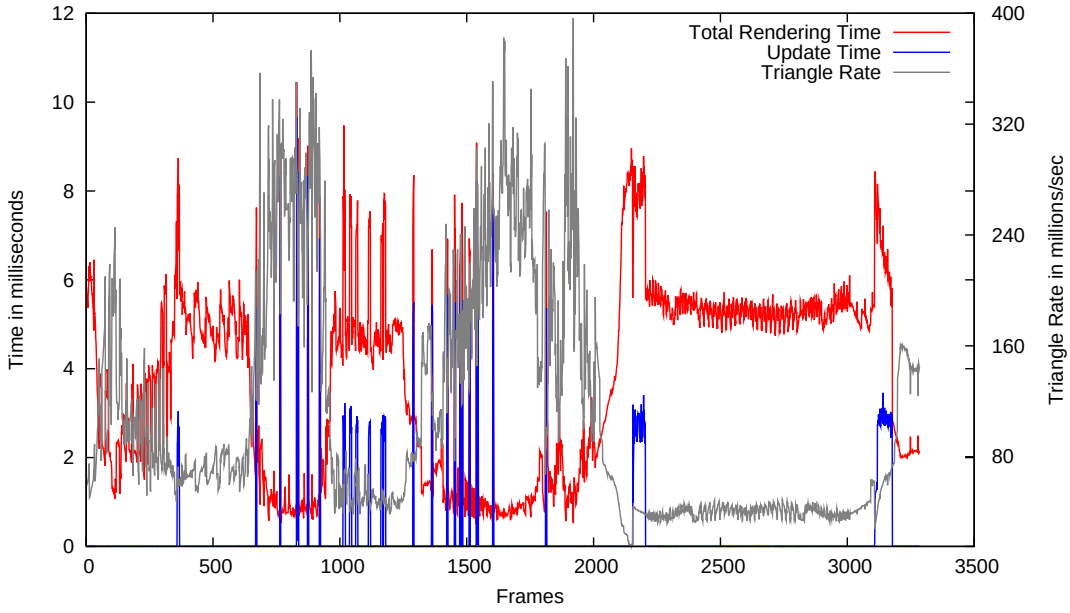

Figure 4.15 Cache update time, total rendering time, and the triangle rate for a typical flight over the terrain.

The rendering time below 8 milliseconds per frame at all times, with the average around 2.5 ms on the trillion sample terrain under different viewer motions. The system can provide a guaranteed 100 fps rate without the CPU, the GPU, or the bandwidth between them being a bottleneck. The system achieves a rendering rate of upto 350 million triangles per second and an average rate of over 160 MT/s. This remarkable rates are made possible by exploiting the power of the GPU. The CPU load stays between 5-10% even when the viewpoint moves up/down. We ran experiments on Puget Sound data using the geometry clipmap demo provided by Hoppe [2] on the same GPU. Their system renders an average of 300K triangles per frame and obtain an average triangle rate of 100 million triangles per second. Our system renders an average of 450K triangles per frame with a peak triangle rate of 350 MT/s.

4.6 Conclusions

In this paper, we presented a system for real-time rendering, deformation, editing, and physics computation of large terrains. The representation enables quick rendering and the ability to manipulate the terrain on-line. The GPU plays the key role in representation, rendering, and manipulation of the terrain. The CPU load is kept very low using the geometry template based rendering, second level culling, and terrain manipulation using fragment shaders. We demonstrate fairly sustained frame rates of over 100 fps and triangle rates of upto 350 million.

The primary limitation of our system is the need for the whole terrain to be present on the CPU memory. This limits the size of the largest terrain that can be handled since data cannot be accessed

from disks at that rate. However, the terrain on the CPU can be thought of as a cache at an appropriate resolution of the terrain that resides on the disk or over the network. A scheme very similar to what is used for the GPU cache can then be used to manage the data on the CPU at an appropriate resolution. Since the CPU cache will need occasional updates, we can update it with a parallel low priority thread using today's dual core processors. The other limitation concerns the speed limit on the viewer imposed by the GPU cache updating. This will improve as the CPU to GPU bandwidth improves on future GPUs.

Chapter 5

Spherical Terrain Rendering

Earth traditionally have been represented by the cartography system using latitudes and longitudes. A 2D grid structure, which we typically see in conventional atlas maps, is wrapped around the sphere using the polar coordinate system and two poles are formed [1]. Terrain rendering is typically done using a 2D array of heightmaps with algorithms optimizing memory usage, CPU usage and quality of rendering [5, 38, 49]. Rendering terrains on sphere seems an easy task because of the coherency between the latitude/longitude system of the sphere and the 2D grid nature of terrain data. The intuitive way would be to use the 2D heightmap data and place them directly at regular latitude and longitude intervals on the sphere. Though the above will apparently solve the problem, it has serious difficulties. The created sphere, shown in Figure 5.1, contains the same number of samples at the poles as at the equator. In other words, the poles get very high resolution and the equator gets lowest. Both these points are indistintive on the surface of the sphere and should get equal detail. Two major problems here are: Uneven sampling of the sphere and inconvenient singularities at the poles where millions of vertices coincide.

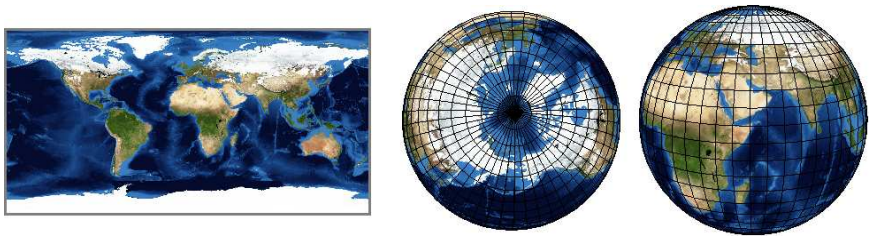

Figure 5.1 Poles have singularity and the whole sphere has uneven sampling.

Another way of sampling a sphere is Hierarchical Triangular Mesh which provides almost uniform detail over the whole surface of the sphere. Starting with an octahedron containing eight equilateral triangles, each triangle is spherically subdivided into four triangles, recursively. The recursion is stopped when a required detail is reached (Figure 5.2). This forms a quad-tree like structure with 8×4^l triangles given l recursions/levels.

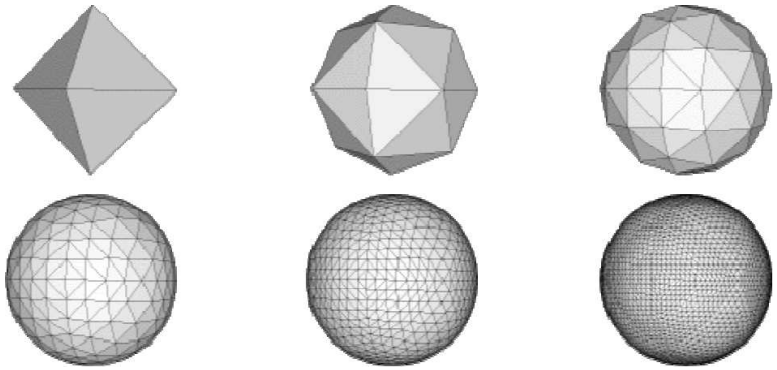

Figure 5.2 After a decided number of recursion a desired detail of sphere is reached *(Image Courtesy: A. Szalay, J. Gray, et al. [48]).*

The HTM, however, is not compatible with terrain rendering techniques. HTMs are fundamentally based on equilateral triangles whereas a terrain renderer uses right-angled triangles to tessellate the regular intervals of heights (Figure 5.3(a)). This difference limits us from directly using a terrain rendering system with HTM structure of the sphere. In this chapter, we present a method in which we adapt a terrain rendering technique to an HTM representation of the sphere. We choose geometry clipmaps as the terrain rendering technique because of its low and constant memory usage, fast rendering and large view ranges [38]. Thanks to HTM, we will get uniform detail, singularity free representation, and clipmaps in addition to fast rendering.

Geometry Clipmaps are based upon creating the mesh with right triangles as discussed earlier. Using the equilateral triangles directly will make the resulting clipmaps rhomboidal in shapes (Figure 5.3(b)) and the relief features will get skewed. We introduce Hexagonal Geometry Clipmaps which pack equilateral triangles perfectly while retaining clipmap's low and constant memory usage and large view ranges. The HTM underneath ensures uniform samples over the surface of the sphere.

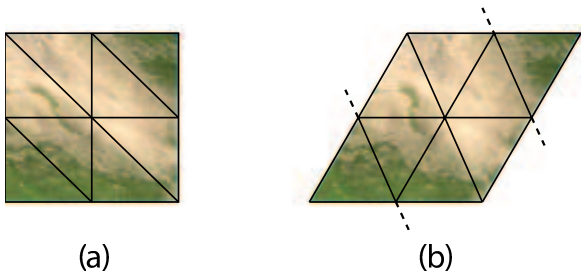

Figure 5.3 (a) Regular terrain, (b) Skewed terrain after samples rendered with equilateral triangles. If clipped from the marked region, yields a hexagon.

5.1 Hexagonal Geometry Clipmaps Overview

Rendering equilateral triangles between the heights of a given terrain data will produce a skewed relief, which is rhombus in shape. We clip the two sharp edges of the rhombus in a way that it forms a regular hexagon as shown in the Figure 5.4. Note that this clipping is used only while rendering; the terrain data resides in square shaped clipmaps always. Following the geometry clipmap's properties, the viewpoint is surrounded by hexagonal shells, each with reduced detail farther from camera. This way, clipmaps can be defined in hexagon shapes to be compliant with equilateral triangles. We describe the process in detail in subsequent sections.

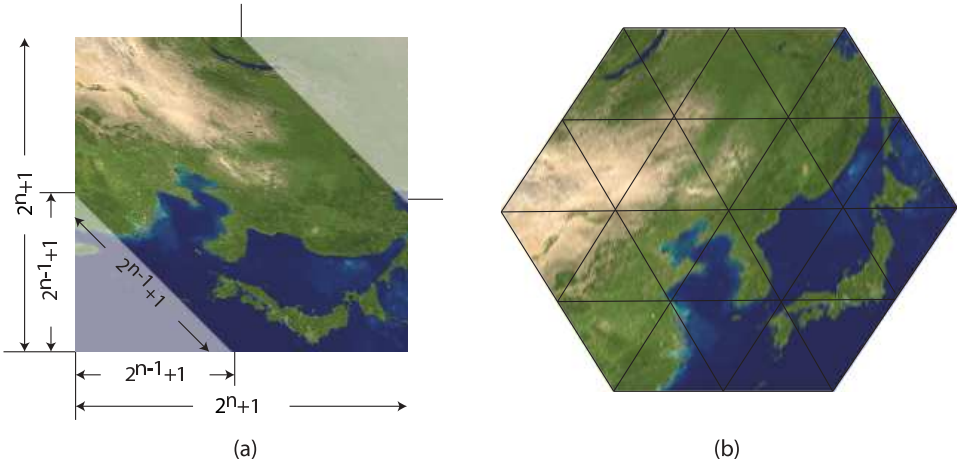

Figure 5.4 (a) 2D grid clipped to form a six sided polygon, (b) The polygon takes shape of hexagon when rendered with equilateral triangles.

5.2 Representation

A hexagonal clipmap uses a 2D clipmap structure for the height data. A 2D clipmap of a size $(2^n + 1) \times (2^n + 1), n \geq 2$ is used in memory. To make a six sided polygon, we diagonally mark $2^{n-1} + 1$ samples at two opposite corner pairs as unusable. In figure 5.4(a), a six sided polygon is shown with $2^n + 1$ samples at each of its sides. If we take three adjacent non-collinear vertices and create an equaliteral triangle between them, the six sided polygon takes the shape of a regular hexagon (Figure 5.4(b)). Note that to form regular hexagons with the same number of samples at each side, we have to maintain clipmaps of size $(2^n + 1) \times (2^n + 1)$ instead of conventional clipmap's size of $(2^n - 1) \times (2^n - 1)$.

Terrain relief features are physically bound with the regular 2D nature of the grid. By using equilateral triangles we skew the relief by 30° in a direction. We transform the heightmap sheared in the other direction as an offline process. This way the original terrain's physical form is retained since the two opposite skews compensate each other.

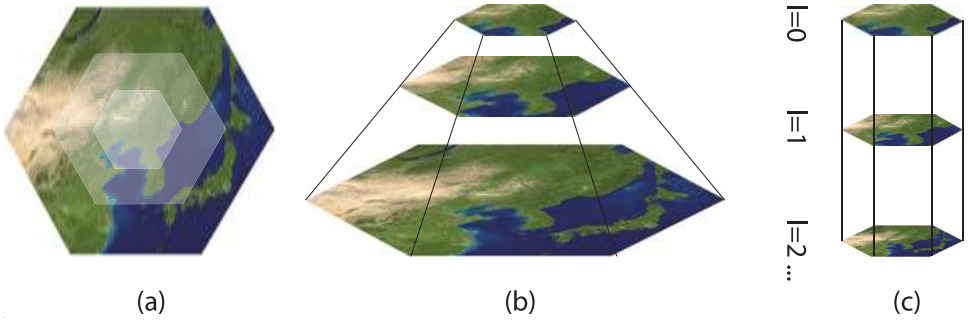

Figure 5.5 (a) Hexagonal clipmaps as viewed from top, (b) Hexagonal clipmaps in the physical form, (c) Hexagonal clipmaps in usable memory.

Using the above representation, we are able to use 2D grid heightmap data, which goes well with the rectangular memory representation of textures on the GPU, with hexagonal clipmaps. Our rest of the representation method follows exactly like geometry clipmaps. We maintain a pyramid stack of clipmaps, with the highest detail given to the middle region and further on reducing detail but occupying more region (Figure 5.5(a)). Memory wise, this takes the form of a cylinderical/cuboidal stack (Figure 5.5(c)); Phyically, this takes the form of a conical shape (Figure 5.5(b)). According to the motion of the camera we update our clipmaps in a way similar to F. Losasso and H. Hoppe [38].

Our representation has the problem of needing extra memory. Since we use hexagonal clipmaps but maintain regular square clipmaps in memory, it leaves some amount of memory unusable which is never used. Hexagonal clipmaps end up using $9/16$ of the square clipmaps loaded in memory. This is acceptable because of the already low usage of memory of clipmaps. Furthermore, we can use those regions to keep terrain data of other parts of clipmap thereby reducing the 2D size of the clipmap. We are investigating this in our future work.

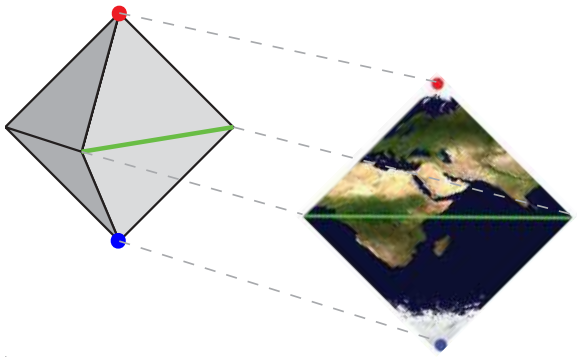

Figure 5.6 Two adjacent base triangles form a diamond. Four such diamonds complete the octahedron.

5.2.1 HTM Terrain Data

An octahedron contains eight equilateral triangles. These *base triangles* are subdivided recursively [48] until the detail of sphere required is reached (Figure 5.2). If we consider any two adjacent base triangles as a single unit, it is a 30° sheared 2D grid of samples. We call such a unit as a *diamond*. We maintain usual 2D heightmap data in memory for these diamonds. The 2D grid data of a diamond has two noticable properties: Part of the cartographic equator aligns with the diagonal of this image and the poles as the two opposite corners (Figure 5.6). This way, a pole is represented by only one sample and equator is represented using the highest resolution. To form a diamond, we pick one base triangle from northern hemisphere and its adjacent southern hemisphere base triangle. Four diamonds form four 2D images and can represent the whole sphere. We consider each of these four 2D images as separate terrains, which are handled independently.

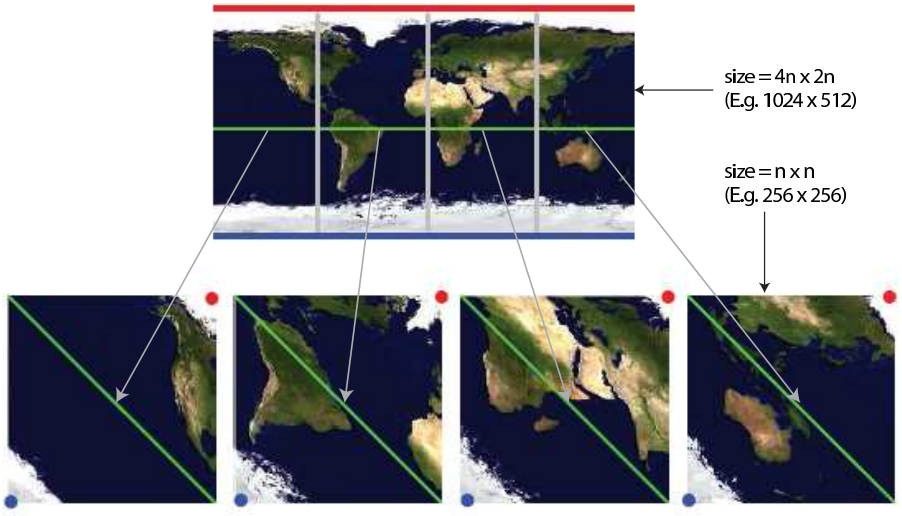

Figure 5.7 A typical planet data converted to be HTM compliant and as a side product it has no redundant information.

5.2.2 Conversion of Planet Data

Planet data is typically available in cartographic form of size $4n \times 2n$ which uses the full row of values to represent the poles as well as the equator. We transform the planet data so that it squeezes into four diamonds. We cut the cartographic map into four images of size $n \times 2n$. Starting from the middle row of pixels in these images, we keep shortening each row till we reach a poles which are left with one pixel each. These rows are aligned as subsequent diagonals in another image which is our diamond and is of size $n \times n$. These four images take $4 \times n \times n = 4n^2$ memory space as opposed to $4n \times 2n = 8n^2$ map size, saving 50% of disk space as all the redundant data is removed (Figure 5.7).

5.3 Clipmap Updates

The terrain data is stored in the main memory in as many resolutions as the number of clipmaps utilized by the system. If we have a terrain of size $N \times N$ heights, and we use l number of clipmaps, then our main memory usage is: $\sum_{x=0}^{l} \frac{N}{2^x} \times \frac{N}{2^x}$. Clipmaps are technically square images and are loaded to the GPU memory as a single array texture. For a clipmap of size $n \times n$, the GPU memory usage is $\sum_{1}^{l} n \times n$. The array texture is bound to a single texture unit and the shader programs have access to all the clipmaps at all times. When the clipmaps need to be updated due to camera motion, the selective layers of the array texture are toroidally updated to hold new data (Figure 5.8). This is an adaptation of the updating scheme of geometry clipmaps of F. Losasso and H. Hoppe [38].

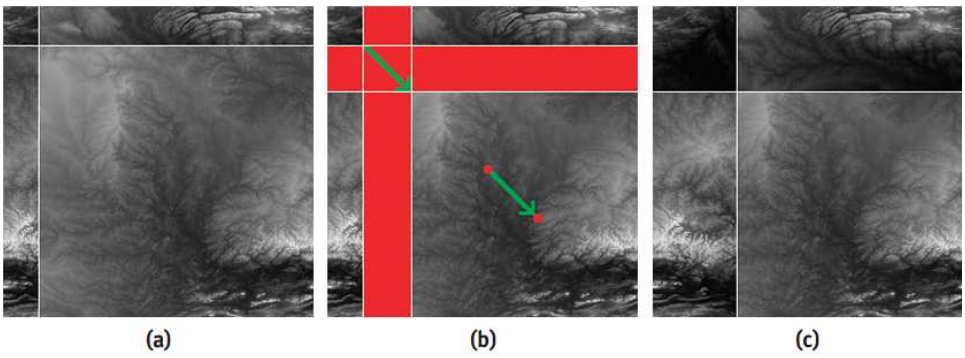

Figure 5.8 With the motion in camera, new data is torroidally updated in the layers of the array texture. (a) Layer before update, (b) Camera moves (green), Update region (red), (c) Layer after update. *(Image Courtesy: A. Asirvatham, H. Hoppe [2]).*

When the camera moves in a horizontal direction, all the clipmaps don't update at the same time. The inner clipmaps require frequent updates than the outer ones. This is due to the difference in resolution. For example, if the camera moves 10 meters, both the inner clipmap and the outer clipmap need to update data for 10 meters, inner clipmap being higher in detail gets more data and outer clipmap gets less data. For camera motion of 1 meter, the outer clipmap may not update at all or will update later. This means, the lower the detail of a clipmap, the lesser updates it needs. This distorts the nested hexagonal grid structure. These *distorted states* periodically cycle as the user moves over the terrain. Even though our memory update mechanism is same as geometry clipmaps, the hexagonal clipmaps' rendering distorts in a more complex manner. Different distorted states need different *rendering blocks* as explained in the next section.

5.4 Rendering

A rendering block in our representation is a parametric vertex buffer object (VBO) containing values in the range $[0, 1]$. Rendering blocks act as templates and can be attached to any given four points (*block-bounds*) to tessellate the enclosing region (Figure 5.9). For a perfect one-to-one mapping of a

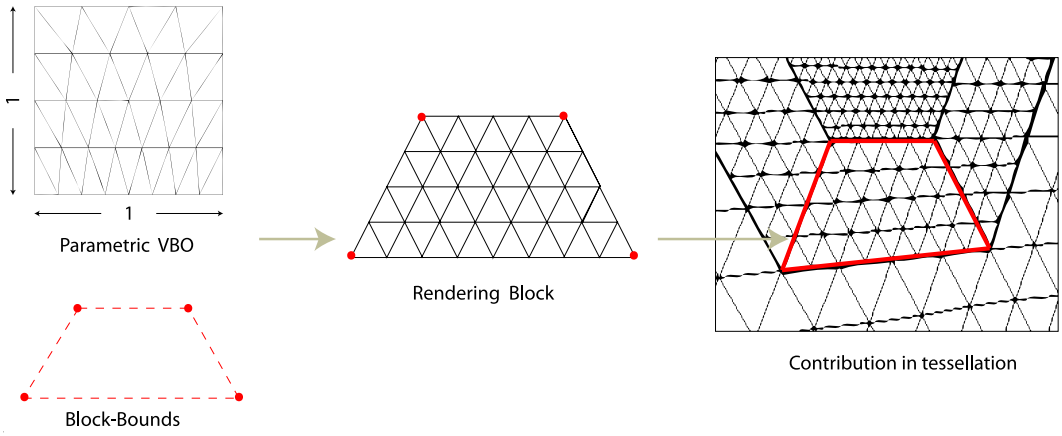

Figure 5.9 Bilinear interpolation at the vertices of a rendering block creates a mesh between the block-bounds.

vertex and a sample in the terrain data, we need to maintain rendering blocks for every unique structure possible coming from the different variations of distortions in clipmap updates (Figure 5.10). A total of 3 unique distorted states are possible and we require 9 unique rendering blocks (an example is shown in Figure 5.9) for a perfect tessellation of the terrain. According to the clipmap size decided by the system, these VBOs are pre-computed and loaded into the GPU memory at the start for continous and quick usage. As the camera moves, the distorted states are calculated between each pair of adjacent clipmaps. For each distorted state, the corresponding VBOs and their block-bounds are sent to the GPU for rendering.

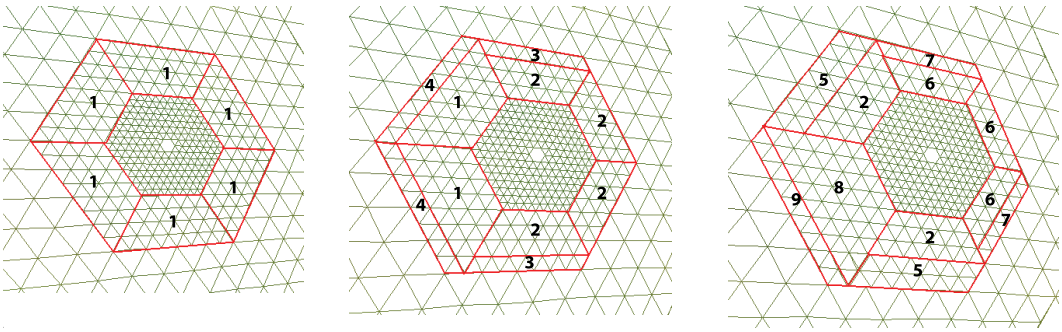

Figure 5.10 As the camera moves these distorted states of the clipmap cycle. Each red quadrilateral comes from a unique rendering block. There are three states and nine rendering blocks are required.

The vertex shader of the GPU gets the parametric vertex from the VBO and the block-bounds. Bilinear interpolation gives the world point p_w on the surface of the sphere. This point is then stretched along the radius R as $P_w = p_w \times (R + h)/R$ to give it the elevation h, which is fetched from the clipmaps kept in the GPU memory as textures. This process does not involve costly operations like sine or cosine to calculate the world point and helps in the performance of the system. However, due to interpola-

tion, the radius of the planet needs to be set carefully according to the resolution of the terrain data and floating point precision limitations.

5.4.1 Smooth Level of Detail Transition

Regions on the planet will transit through low to high resolution or vice versa when the user is continously moving on the planet. The height data will change due to the resolution shift which happens when it goes through different clipmaps. This can create sudden jumps in the rendered terrain and can reduce the rendering quality. We morph the heights of the clipmaps with its lowest resolution version, weighted by the distance from camera by the following formula:

$$\alpha = \max(\min((ax_1 + by_1 + c)/W, 1), 0)$$

$$h = h_l(1 - \alpha) + h_h\alpha$$

Where α is morphing factor, W the transition distance, (x_1, y_1) the texture coordinates at the elevation point, h_l the elevation at low detail clipmap, h_h the elevation at high detail clipmap and $ax + by + c = 0$ the line equation of a clipmap's side in texture space. Note that the line equations' orientation are fixed for sides, the hexagon never changes its orientation since it has to fit the HTM triangles. Because of this, (a, b, c) are constants for each of the six sides. While the camera moves, this changes the height slowly to the low resolution (Figure 5.19), so that when the next clipmap comes, the height data is already changed and we don't see any quirks in the visuals. This calculation is done in the vertex shader. The rendered terrain is technically dynamic but looks rigid with the motion.

5.4.2 View Frustum Culling

At any point of time, the camera sits at the center of a giant hexagonal space. By getting the camera yaw angle, we select the triangular areas of the hexagon which are in the field of view. Given a horizontal field of view of $90°$, three adjacent triangular regions are marked for rendering (Figure 5.11). Rendering blocks are not triggered for the unmarked triangular regions. This selection is done at the begining of every frame to save unnecessary computations.

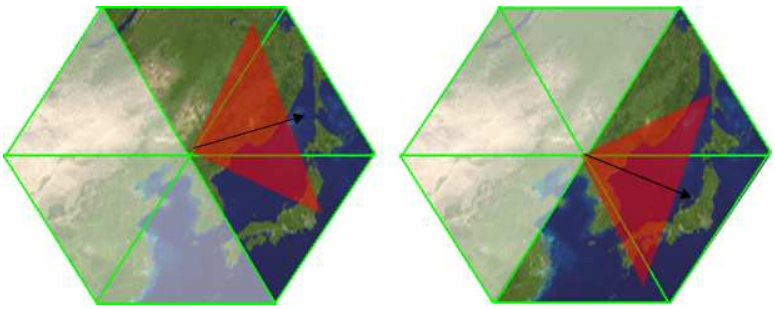

Figure 5.11 View Frustum Culling is a mere selection of relevant sides according to camera yaw.

5.4.3 Handling the Whole Planet

Each diamond is handled as an independent terrain. To handle the whole spherical planet, we arrange four diamonds in a *big-texture* (Figure 5.12). Clipmaps wrap at the bounds of the big-texture. Most of the time the camera hovers around a single diamond and nothing special needs to be done to handle the whole planet. But there are two more cases. A camera travelling across the longitudes will have moments when two diamonds are visible. When the camera is near the poles, terrain spans over four diamonds (Figure 5.13).

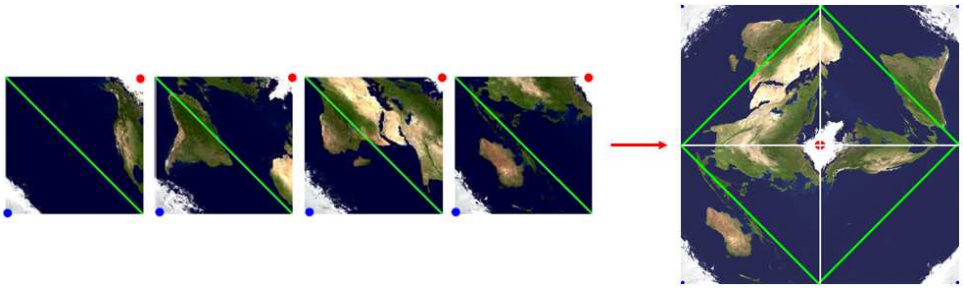

Figure 5.12 Diamonds combined into a single big-texture. Note that the North Pole comes in the middle and South Pole is at the corners.

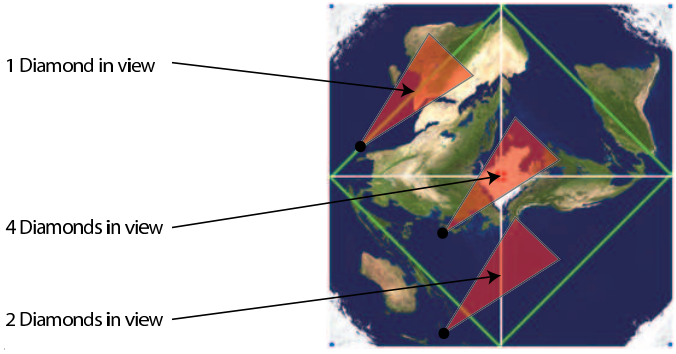

Figure 5.13 Movement of camera accross the whole planet will encounter multiple diamonds in view. At the poles, camera will see a terrain which is spanning over all the four diamonds.

In such cases, diamonds are still treated as independent terrains. That means, if four diamonds are visible, each will be rendered four times using its clipmap data. The rendering blocks going out of bound of the diamond are culled. Few rendering blocks are partially out of bounds, for those, the out of bound triangles are *buried* by making their elevation negative or zero. Note that the big-texture theoritically carries the whole dataset. If the dataset is huge, the big-texture may need to be kept in parts.

5.5 Results

Hexagonal clipmaps give optimal rendering throughput and steady rendering similar to square clipmaps. Distances from the edges of the hexagonal clipmaps to the viewpoint vary less than square clipmaps, since hexagon is closer to a circle than a square (Figure 5.17), it provides better uniformity in triangle count on the screen at any camera yaw angle. We implement our system in OpenGL 2.1 and tested on Nvidia GeForce 8800 Ultra, GTX 280 and GTX 480. For the purpose of the experiments, we took the Puget Sound data and used it for each diamond over the HTM sphere. For continuity between the diamonds, we flipped the data to be continous. We use a clipmap size of 257×257 forming a hexagon with side of 129 samples. We get steady framerates in a typical walkthrough over the terrain. Figures 5.14,5.15,5.16 show our performance results on puget sound data. Some rendering results can be seen in Figures 5.17,5.18,5.20.

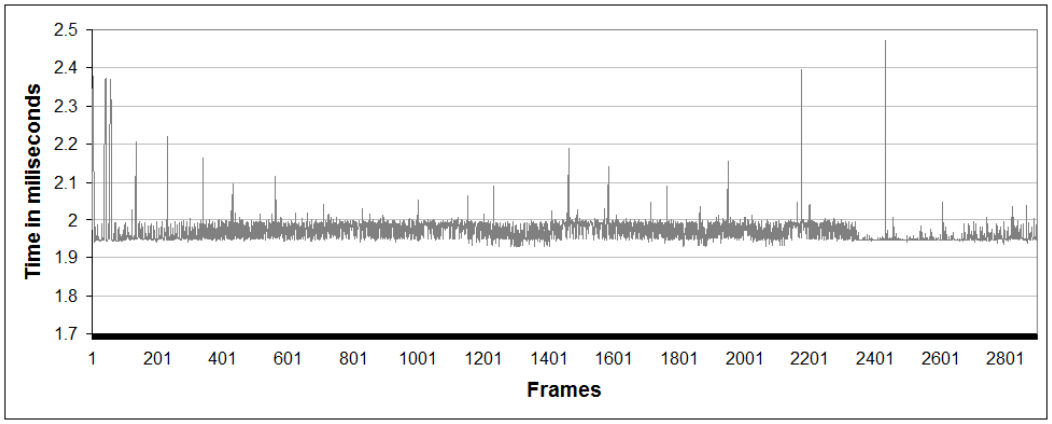

Figure 5.14 Performance resuts on Puget Sound data treated as a diamond, on an Nvidia 8800 Ultra.

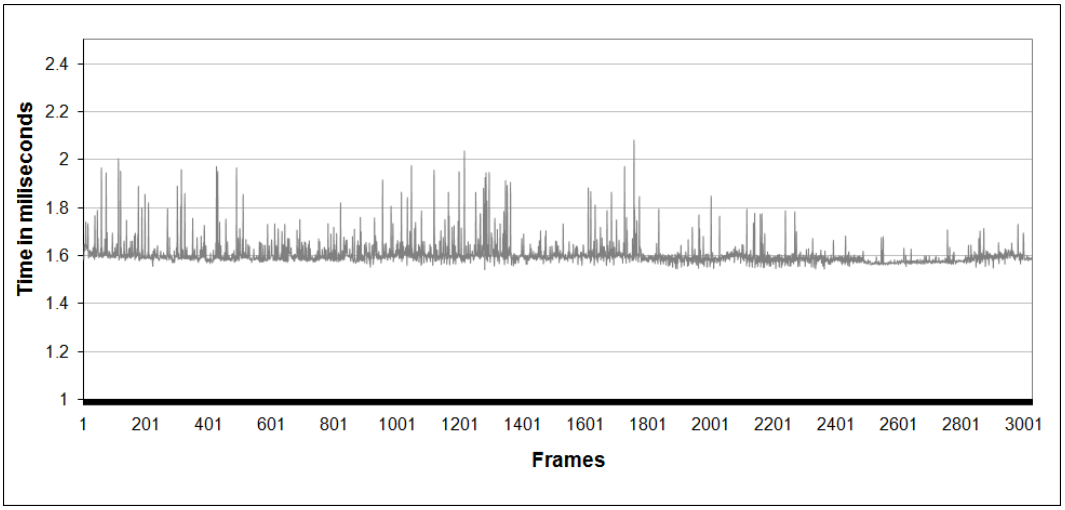

Figure 5.15 Performance resuts on Puget Sound data treated as a diamond, on an Nvidia GTX 280.

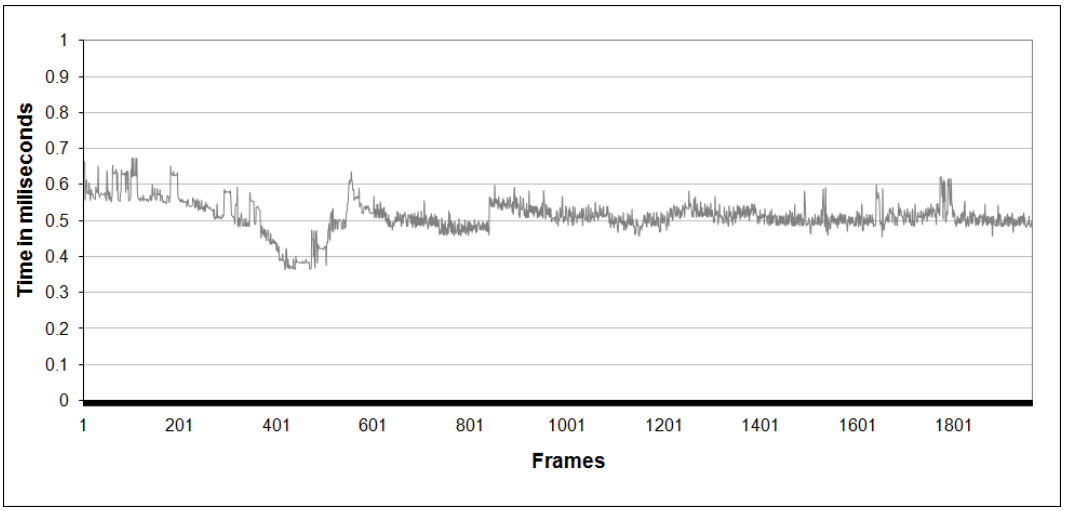

Figure 5.16 Performance resuts on Puget Sound data treated as a diamond, on an Nvidia GTX 480.

5.6 Conclusions

We presented a Spherical Terrain Rendering algorithm which provides uniform sampling of points over the surface and fast rendering with low memory usage. Hexagonal Clipmap provides the best two methods: HTM is best to represent spheres and clipmaps are best to render terrains. Applications like Google Earth/Virtual Earth, space simulators, 3D social networks (e.g. Second Life) or spacecraft involving games can show seemless journey from ground to space using this method. In future, we would like to improve our memory usage and try to fit usuable terrain data in unusable regions of our clipmaps on the GPU. Apart from this, HTM primarily being a geospatial indexing method, we would incorporate searches in different resolutions with addressing and fast look-up.

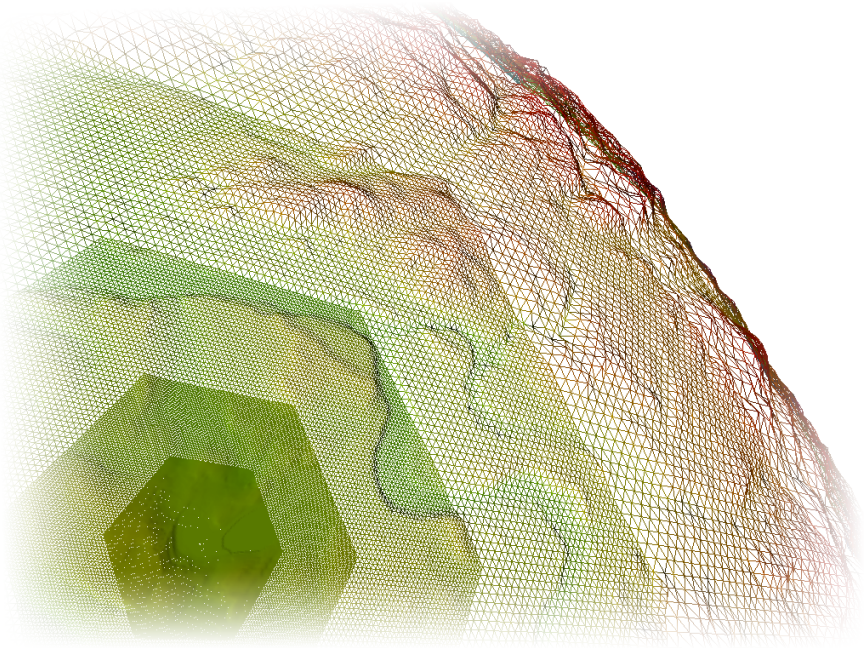

Figure 5.17 Hexagonal Clipmaps are closer to the shape of concentric circles.

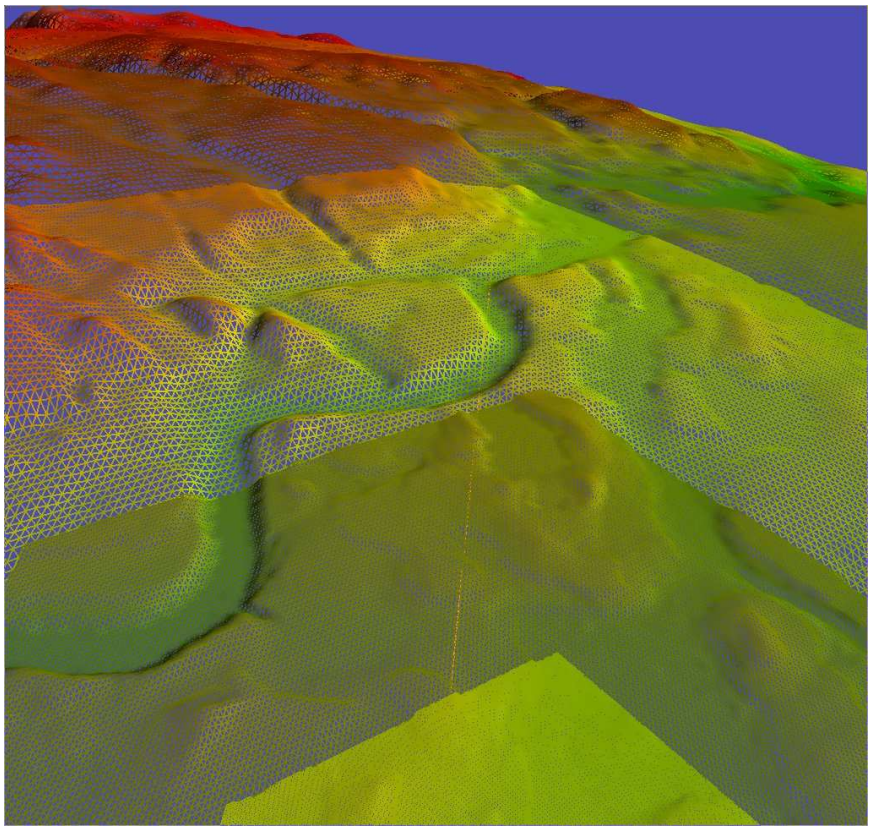

Figure 5.18 Different clipmaps shown in different shades.

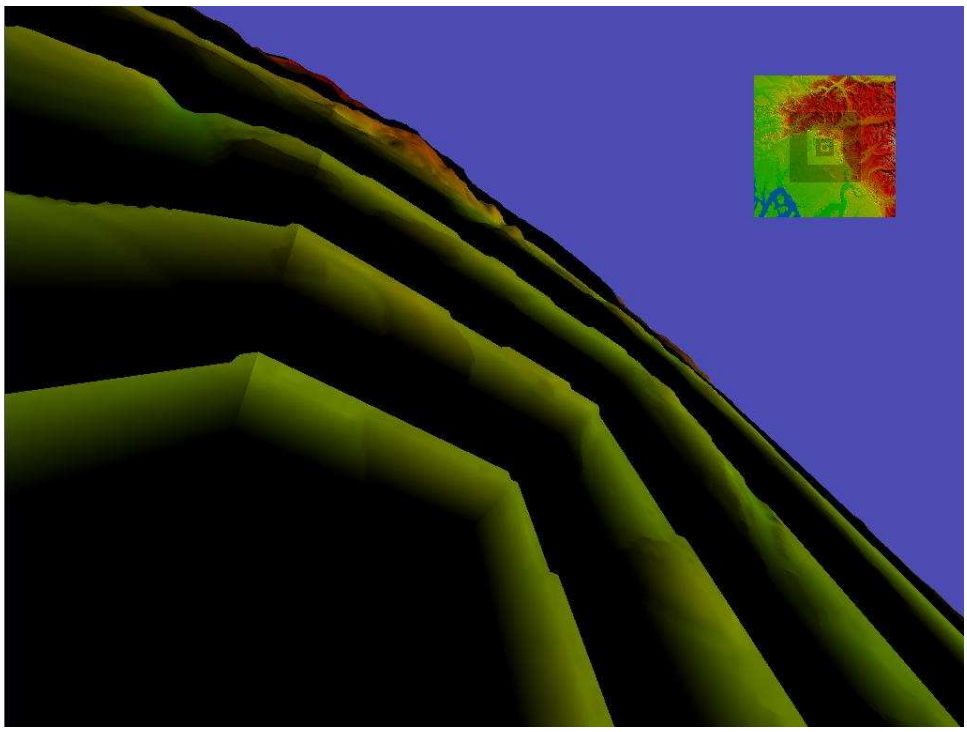

Figure 5.19 Blending factor in transition, blackness level indicates value of α.

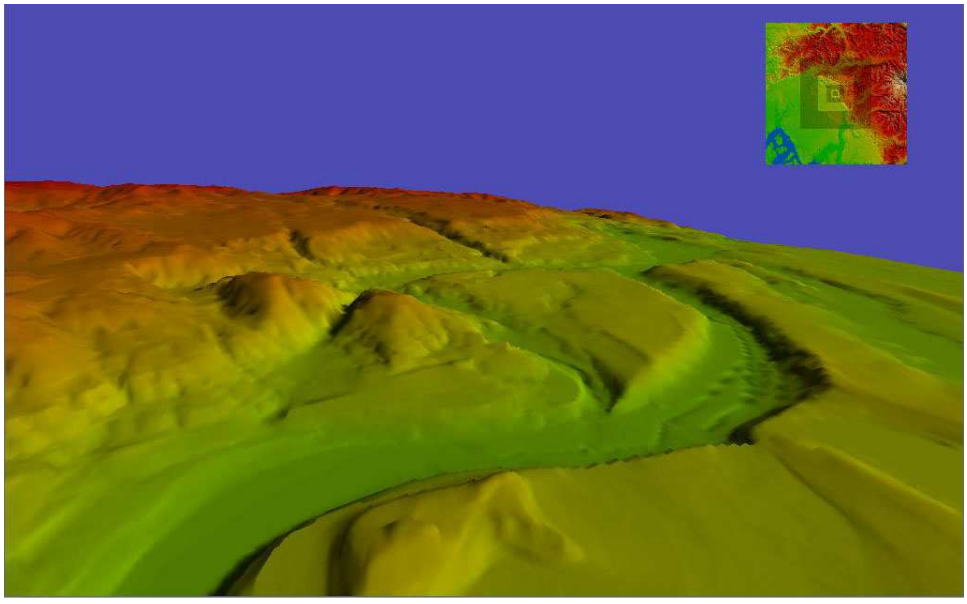

Figure 5.20 A scene while a flythrough on the spherical terrain.

Chapter 6

Painterly Rendering of Terrains

The regular nature of the terrain data make them a specific type of model. We exploit this special nature of terrains to provide efficient painterly rendering for them. A technique to order the triangles of a terrain from back to the front is at the heart of this. We achieve an fps of 120 on Puget Sound terrain data on the Nvidia 8800GTX GPU. In this chapter, we present a real-time painterly rendering technique to make abstractions of terrains. We also emphasise our results with post processing for varied stylizations.

We built our painterly rendering system over the terrain rendering system explained in Chapter which can achieve 150 fps with an average rate of 84 million triangles per second and a highest of 200 million triangles per second on current GPUs.

The organization of the chapter is as follows: A brief overview of the system is mentioned in section 6.1. In section 6.2 we show the representation of terrain data and stroke textures. Here we also explain view frustum culling and level of detail management. section 6.3 shows the method in which we are ordering the strokes in back to front order. Technique for rendering the strokes is mentioned in section 6.4. Illustrations and the performance of our system are discussed in section 6.5. We conclude with a discussion on technical aspects and aesthetic considerations with some future works in section 6.6.

6.1 Overview of our Approach

Terrains are heavy objects, often involving millions of triangles in each frame. Conventional two-pass painterly rendering techniques will be inefficient for them. We combine painterly rendering with terrain rendering optimally for real-time performance. We treat each height in the elevation map of the terrain as a stroke's location in the 3D world. The point location is projected on 2D screen using projection transformation and a rectangular stroke is rendered at that location, orientated along the projected slope of the terrain (see Figure 6.1). Real-time performance is obtained using the following.

1. Only the strokes of the visible part of terrain are rendered for efficiency. This is achieved with a view frustum culling algorithm.

2. The strokes are rendered in a back to front order for alpha compositing. We exploit the special property of terrain representation to obtain the back to front ordering in one pass. This is explained in section 6.3.

3. The level of detail of the terrain is changed smoothly with distance from the viewpoint. This avoids the problem of strokes getting cluttered at far distances, which can be visually distracting. Level of detail also reduces the rendering load.

The whole terrain is kept in the CPU memory. A section of it needed for rendering is cached on the GPU memory as elevation maps. Precomputed stroke texture, color texture, normal map, and the slope map are also stored in the GPU's texture memory. The terrains are cached in terms of 1024×1024 blocks and are rendered in terms of 64×64 tiles. The tile is the basic unit for rendering, view frustum culling, and LoD management.

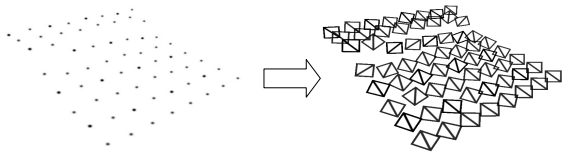

Figure 6.1 Each height in the height-map is converted into a rectangle which is oriented along the terrain's slope at that point. An 8×8 grid is shown as example.

Each stroke is sent by the CPU as a single point primitive as a geometry template, which gets converted into a rectangle on which a stroke texture is mapped. This is accomplished with DirectX10/SM4.0 based shaders explained in section 6.4. Each point on the terrain is rendered as a stroke. The stroke is aligned in the direction of the slope at the 3D terrain point to imitate how artists draw such scenes. We render the strokes in the back-to-front order by exploiting the regular grid structure of tiled terrains. Points of a tile can be scanned and rendered as strokes in the back-to-front order, based on the view orientation. Eight such orderings are sufficient to handle any view orientation. The tiles that survive frustum culling are also rendered in the same order to provide a back to front ordering for the entire terrain without sorting. This procedure enables us to render arbitrarily large terrains at frame rates of 120 and above in the painterly style.

6.2 Terrain Representation

Our base terrains are 2D grids of heights with a fixed post-distance in the X and Y directions.Our focus is on painterly rendering of the terrain at real-time rates without the CPU, the GPU, or the bandwidth between them becoming the bottleneck. The available terrain data is loaded in the CPU memory and a contiguous window of the terrain is kept in the video RAM of the GPU based on the current viewpoint.

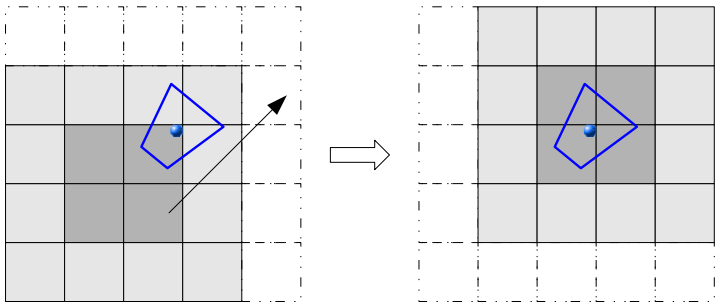

Figure 6.2 Reference point is at the center of ground-plane projection of the view frustum (marked as blue). Reference point is kept within the 2×2 blocks. As it goes out it is re-centered. The figure assumes 4×4 cache size.

6.2.1 Representation of data

Terrains are divided into fixed memory-size *blocks*, each of which is divided into a number of *tiles*. A tile is the basic rendering unit for the CPU. Currently, blocks are of size 1024×1024 and tiles of size 64×64. These blocks are loaded as textures on the GPU memory. We maintain a *GPU cache* consisting of $N \times N$ blocks which gets updated periodically to hold all the data needed for rendering. We try to keep the GPU cache symmetric with respect to the projection of the view frustum on an average "ground" plane. We do that with the use of a *reference point* which is kept close to the center of the GPU cache (Figure 6.2). We use the center of ground-plane image of the view frustum as the reference point currently. This ensures fixed in memory representation for the terrain.

If the reference point goes beyond the central 2×2 block of the GPU cache, the cache is re-centered by bringing another row or column of blocks (Figure 6.2). Since the cache is maintained in memory as an array of texture ids, re-centering involves downloading a few blocks to the GPU and adjusting pointers on the CPU. The data transfer time is kept small using a job-queuing scheme. The blocks to be brought in the GPU cache are not done at once, but done successively in following frames to avoid possible jerks. The basic terrain system is able to render large, CPU resident terrains at above 100 fps along with the cache updating in the background.

6.2.2 Level of Detail

The view frustum culling algorithm treats each tile as basic units. The bounding sphere of tiles are tested against the six planes of the frustum. On the basis of this, tiles are marked to be inside or totally outside the frustum, and are assigned with a *LOD* number. LODs (Levels of detail) for a tile include different resolutions of an area on the ground. A particular LOD of a tile can be computed by dropping alternate samples from the better LOD available. Highest LOD for a tile contains all the samples. We calculate the rendering LOD of a tile using its distance from the viewpoint (Figure 6.3). Farther the distance, lower the LOD. LOD l becomes a function of distance d as the integer part of

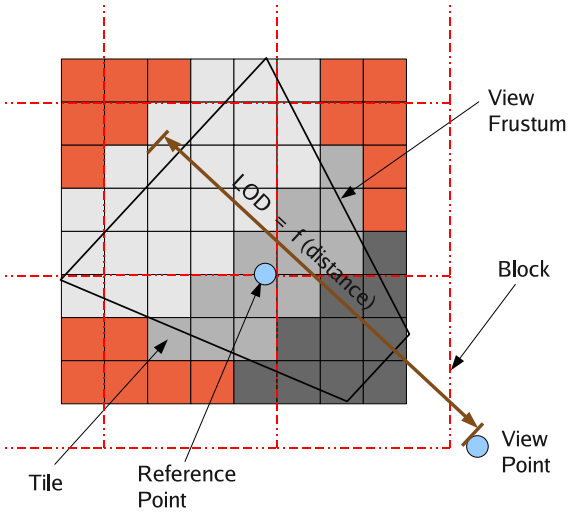

Figure 6.3 Tiles outside view frustum (marked red) are eliminated. Tiles totally inside (grey shaded) are rendered with strokes at each of its sample's locations. LODs of tiles to be rendered and the blending factor is calculated as a function of distance. Fewer strokes are drawn for a lower LOD tile.

$l = \log(1 + d/d_t)$, where d_t is a pre-decided LOD transition distance. When the LOD of a tile changes from one to another, many samples/strokes may pop up suddenly. For this, we morph the tile from one LOD to other by fading the alternative strokes away as they go out and vice versa. The fractional part of l is used as the *morphing factor* and is multiplied to the opacity of alternative strokes in the vertex shader. While Wagner [49] uses the morphing factor to geomorph two different heights at that same location, we use it to fade in or fade out the strokes which are coming in and going out respectively, giving a smooth transition without popping artifacts.

6.3 Back-to-Front Stroke Ordering

A back-to-front ordering of samples/strokes of the terrain is at the heart of our algorithm. We discretize the camera yaw into 8 zones of each $45\,\mathrm{deg}$ each shown in Figure 6.4(a). Each zone corresponds to a particular order of scanning the heights for guaranteed back-to-front ordering of triangles. The 8 zones have unique ordering, four of which are shown in Figure 6.4(b). The same scan order applies to the tiles inside the view frustum as seen in Figure 6.5. In practice, we switch the ordering a little while after the viewpoint is into the new zone to avoid unnecessary toggling of the ordering at the boundaries between zones. Tiles are rendered as *VBOs* (vertex buffer objects) for good performance. A single VBO can render any tile, as other parameters like tile's world origin, blending factor etc. is packed up in texture coordinates. For a given range of orientation of the camera, an ordering is fixed. Thus each zone corresponds to a unique VBO.

The same order is used to scan the tiles for view frustum culling. Figure 6.5 shows one out of eight of the possibilities for tile scanning shown in Figure 6.4(a). All the tiles farther from the camera get

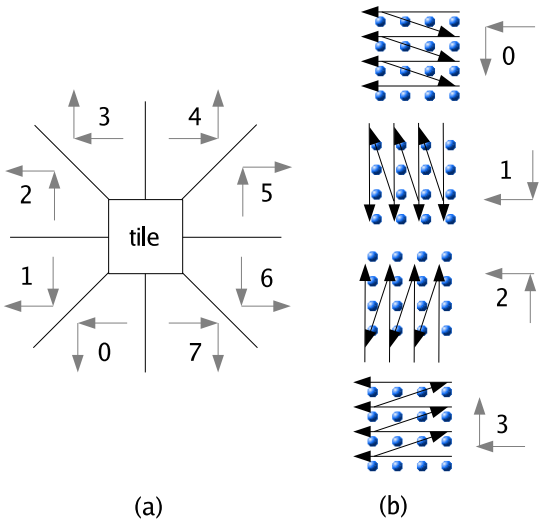

Figure 6.4 (a) A tile can be viewed from many yaw directions, but only eight zones are sufficient for a back to front ordering of samples in it. (b) Four possible arrangements of samples for some ranges shown in (a); Other ranges can be handled in the similar way.

rendered before the nearer ones. Because of this, all the strokes in the screen in that view become ordered from back to front without the cumbersome need of sorting.

6.4 Stroke Rendering

We send points to the graphics pipeline for each stroke to be rendered. Vertex shader computes the exact world location of the stroke at this point. It also calculates the color from the texture and normal map of the terrain with other lighting information (the unified architecture of latest GPUs allow fast

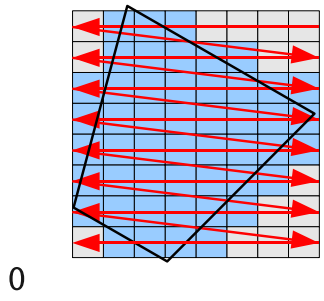

Figure 6.5 View frustum culling algorithm testing tiles in a specific order depending upon the camera's orientation. Here zone 0 is shown. Such eight orders of testing are possible as explained in Figure 6.4.

texture access from any shader [7]). The alpha of the point is changed according to the morphing factor decided for that tile from the CPU. The vertex shader forwards these things to the pipeline.

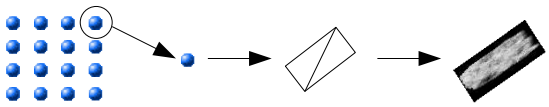

Figure 6.6 Overview of rendering of stroke. Each vertex from the VBO gets converted into a rectangle which is mapped with a stroke texture.

Geometry Shader of the GPU can generate primitives [7]. It converts the single point primitive sent from the CPU into a rectangle for the brush sprite (Figure 6.6). The perspective division of the graphics pipeline makes the strokes smaller when they go farther, while painterly rendering needs constant sized strokes. To compensate for this process, the vertices are multiplied with the w value (the perspective scale factor) before rasterization. This reverses the division (Haller and Sperl [20]) and the strokes always maintain the same size on the screen. This process can lead to holes in the surface if the camera goes very close to the ground for a given point density. We disable the multiplication at such distances when the strokes start to lose density.

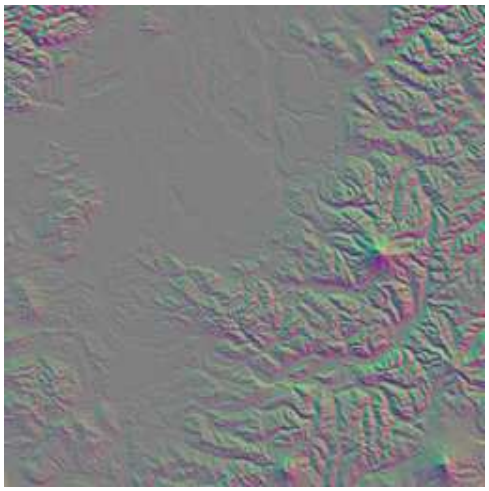

Figure 6.7 Slope-map, Puget Sound dataset

The generated rectangle is subsequently oriented in screen space along the slope of the terrain at that location since artists tend to place their strokes along the slopes of mountains running down to the valleys. We precompute a *slope-map* that gives the direction of maximum gradient at every point in the terrain (Figure 6.7). Slope-map stores the gradient vector in the world space, which is accessed by the Geometry Shader for every sample, is transformed to camera coordinates and to the image space to get the stroke orientation.

The fragment shader accesses the stroke texture, and modulates its color with the color coming in from the pipeline. Alpha blending happens between these rendered strokes so that they mix among themselves for a smooth output. The outline of the whole method is described in Algorithm 2.

Algorithm 2 Painterly Rendering of Terrains

1: Load stroke textures st
2: Load height H, color C, normal N, slope S map of terrain
3: Create 8 VBOs for different camera yaw-ranges
4: **for** each frame **do**
5: Update GPU Cache if necessary (section 6.2.1)
6: Determine zone q depending on the yaw-range of the camera
7: Perform VFC and LOD assignment based on q.
8: **for** each tile **do**
9: Send VBO[q]
10: **Vertex Shader:** Calculate color using lighting
$c = f(C, N)$. Calculate position p using height H
11: **Geometry Shader:** Generate a quad at p,
orient along slope S, assign color c
12: **Fragment Shader:** Output color $c_o = mix(c, c_{st})$.
At a different render target, output color as normal of stroke texture N_{st}
13: **end for**
14: Phong shade the output using the normal map
15: **end for**

For more stylizations, we render the normal maps of these strokes separately as well. We do this with multiple render targets supported by modern GPUs. In a different pass, these two outputs are treated as a texture and its normal map respectively, and are mapped on a screen aligned quad. With the help of the normal map, the scene can be Phong shaded with a varying lighting source (Figure 6.8). This process is inspired by [24] but we do it in real-time on rendered outputs harnessing the power of modern GPUs.

6.5 Results

We built our system and experimented on a Intel Pentium Core 2 Duo E6400 as the CPU and an NVIDIA 8800GT as the GPU. We used the OpenGL 2.1 graphics library and GLSL 1.20 shaders. We chose different screen resolutions to render upon for speed of alpha blending is screen size dependent. Performance is dependent on stroke size as well. We choose an optimal stroke size; Small enough to give good performance but not as small to leave holes in the terrain. With a resolution of 1024×768, we got seamless performance with an average triangle rate of 40 million triangles per second (Figure 6.9). With a resolution of 1280×1024 we get 35 million triangles per second and 120 fps (average). Traditional two pass painterly rendering technique (with depth map computed in the first pass) had half the performance of our system. We did our experiments on Puget Sound terrain data available from Georgia Tech website. Blue marble data set was also included in our experiments. We used some real satellite terrain data-

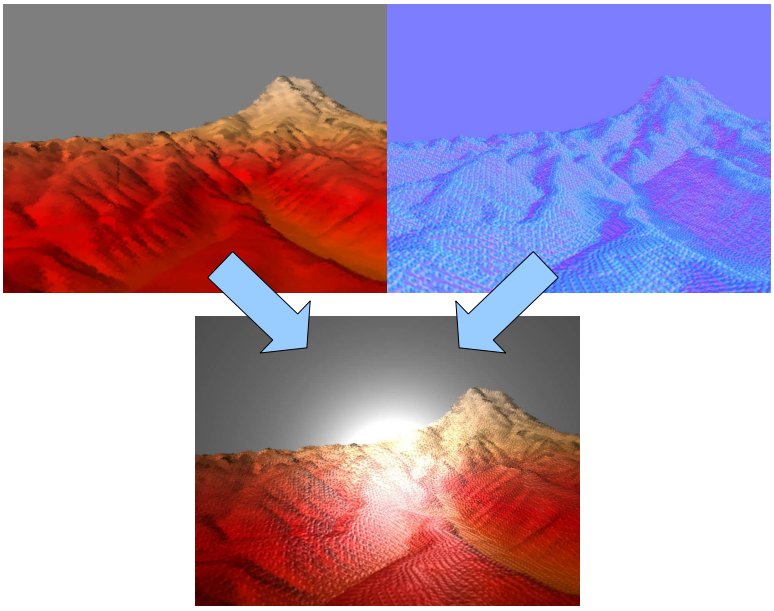

Figure 6.8 The color output and the normal map output of the scene are used to Phong shade on top of it to stylize it. The effect is that of shining a spotlight on the painting. The normal map is contrast stretched here for visibility.

set and some synthetically created ones as well. We show the effects of different stroke directions, with along the slope direction. In Figure 6.12 and 6.10(top-right), the strokes are oriented along a perpendicular direction to the XY projection of the normal vector. This simulates the effect of strokes flowing over the ridges instead of along the slopes. An artist drawing with strokes of fixed orientation is shown in Figure 6.10(bottom-left). Effect of adding small randomess to orientations is shown in Figure 6.10(top-left). Figure 6.10(bottom-right) shows the use of a small brush with sharp strokes. The accompanying video contains painterly walk-through on Puget Sound data. Some of our results are shown in Figure 6.11, 6.13, 6.14, 6.15, 6.17, 6.16.

6.6 Conclusions

We presented a real-time painterly rendering technique for terrains. We get nice visuals with frame to frame coherence on animation of the scene. Considering rich nature of terrains and cumbersome nature of painterly rendering processes, we get good performance with our system using latest graphics hardware. Our system being single pass only, is faster than traditional painterly rendering techniques involving two passes. With varied stroke textures, and orientations of strokes, different artistic styles can be achieved with variety of taste. In future, we wish to render terrains with procedural stroke textures similar to geo-graftals mentioned in [39] and [30] to create even varied visuals and improve performance by optimizing the techniques specifically for terrains.

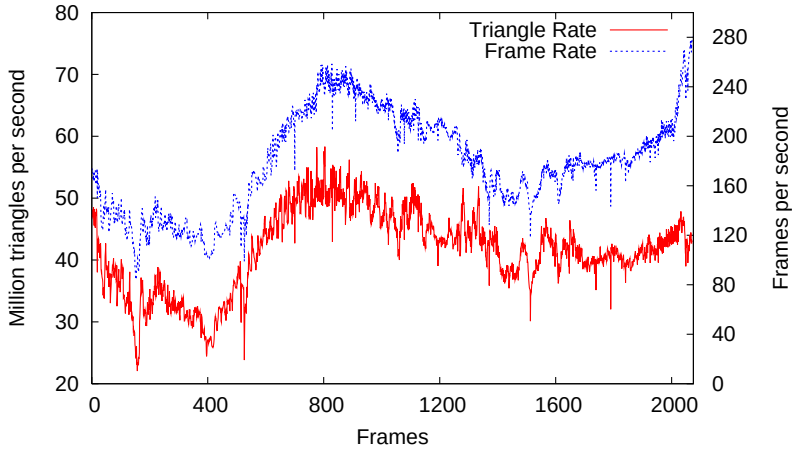

Figure 6.9 Walkthrough over the terrain

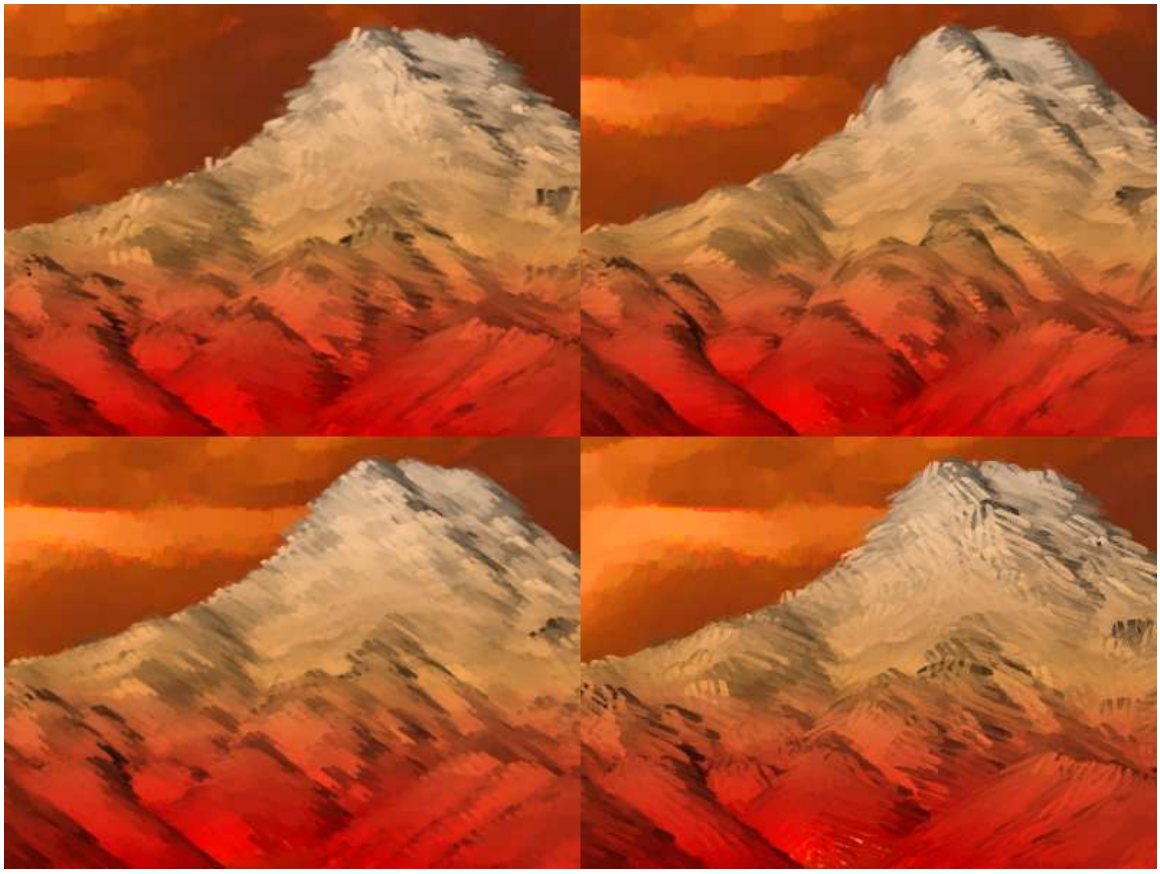

Figure 6.10 (top-left) Strokes placed along slope with some perturbations in orientation. (top-right) Strokes placed along the perpendicular to the normal. (bottom-left) Strokes placed with a fixed orientation. (bottom-right) A sharp stroke texture. Sky is a pre-painted texture.

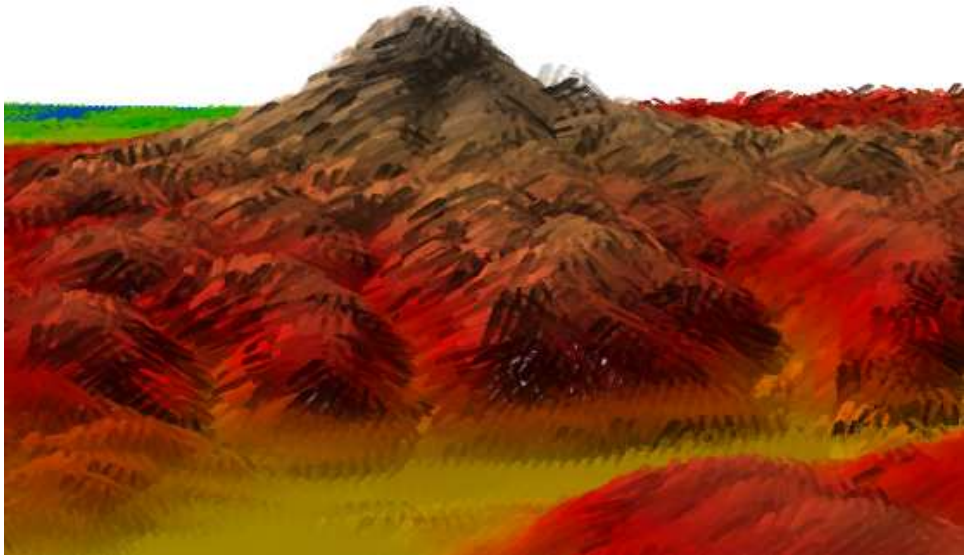

Figure 6.11 Distant view of Mount Rainer

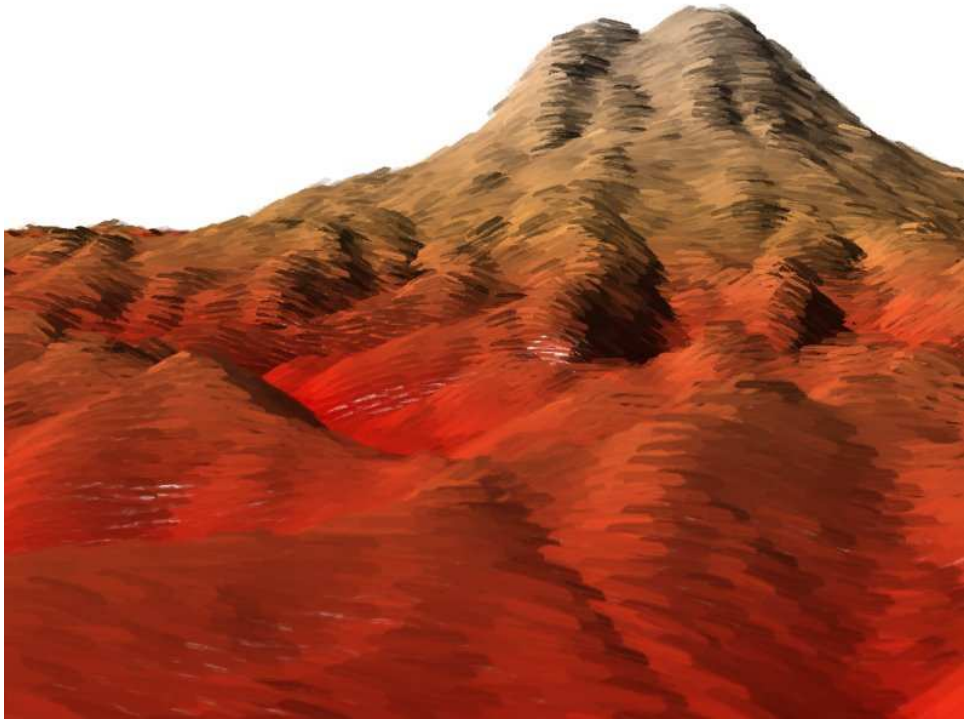

Figure 6.12 Strokes running along perpendicular to normals.

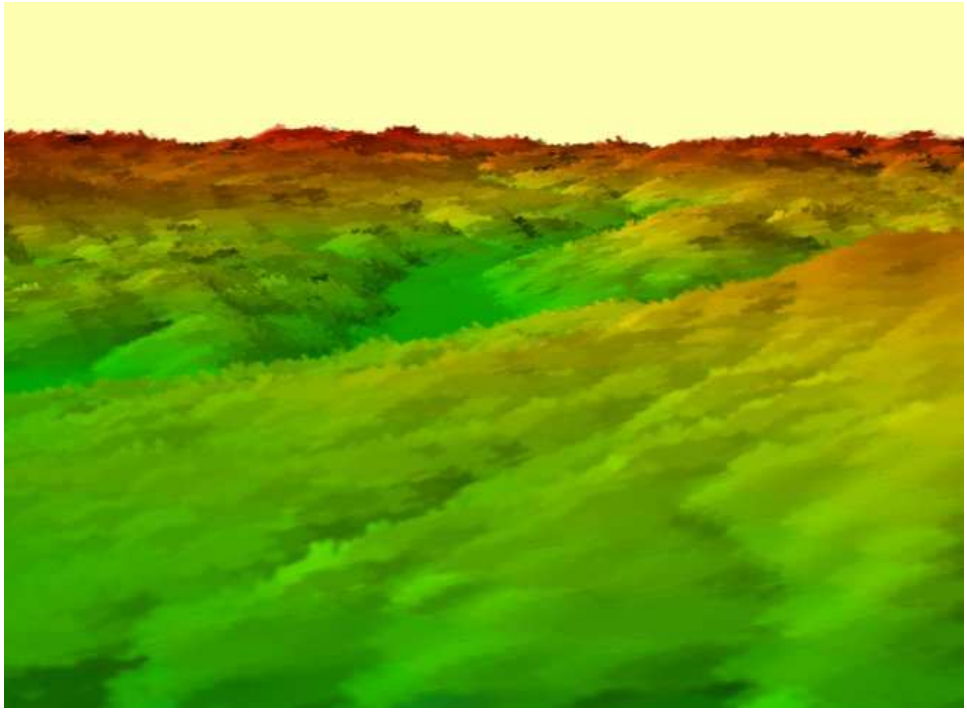

Figure 6.13 A region in Puget Sound painterly rendered which has low height variations.

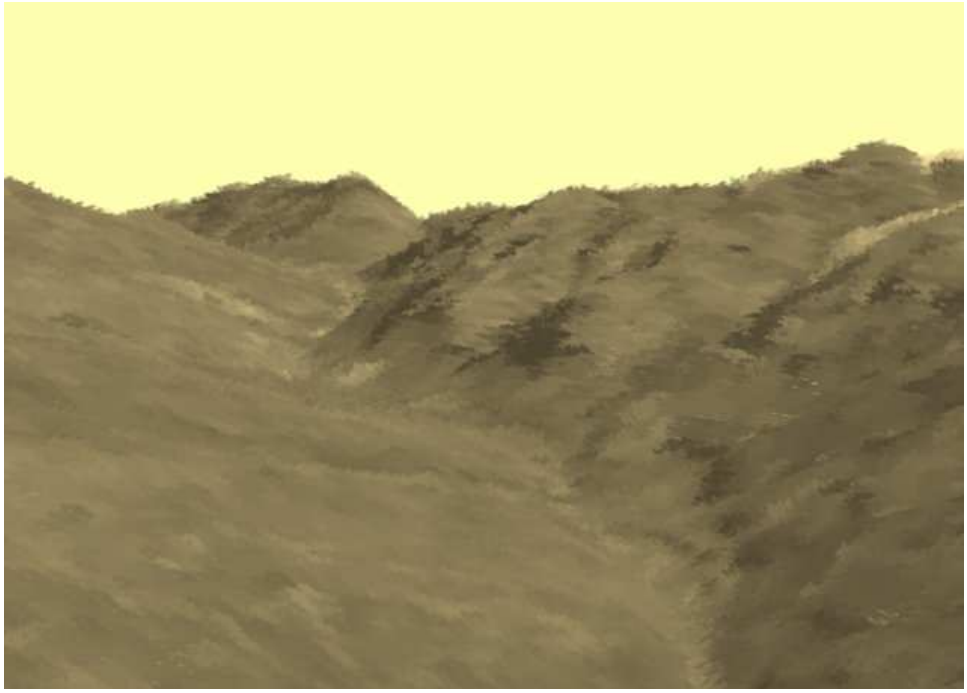

Figure 6.14 A real textured dataset rendered in a painterly style.

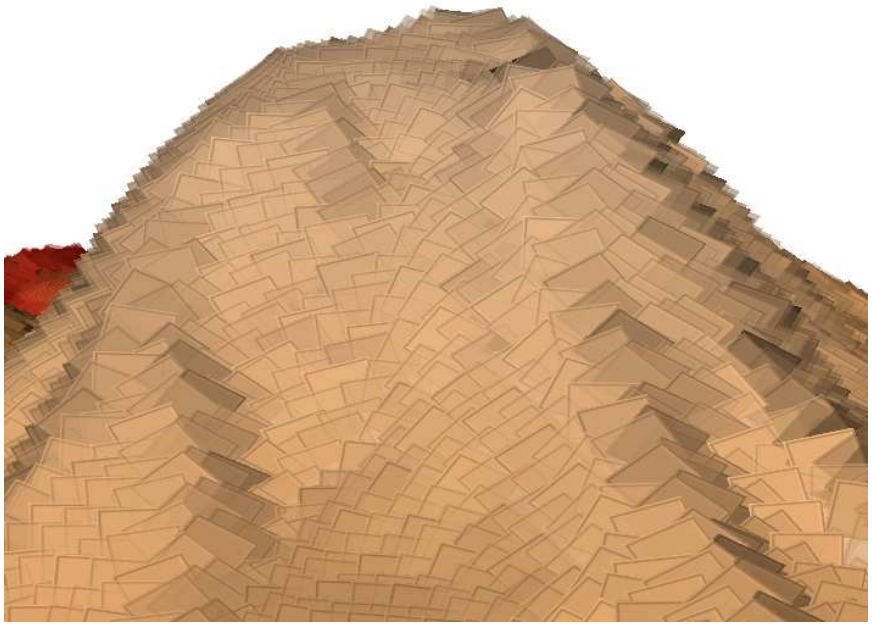

Figure 6.15 Simple rectangles are used instead of proper brush strokes to illustrate the flow of strokes along a hill

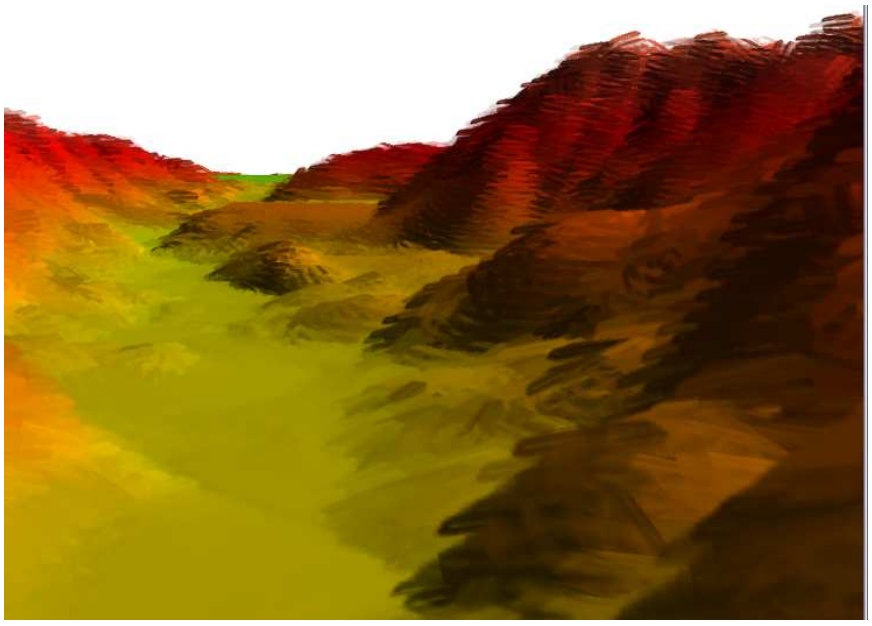

Figure 6.16 A valley region

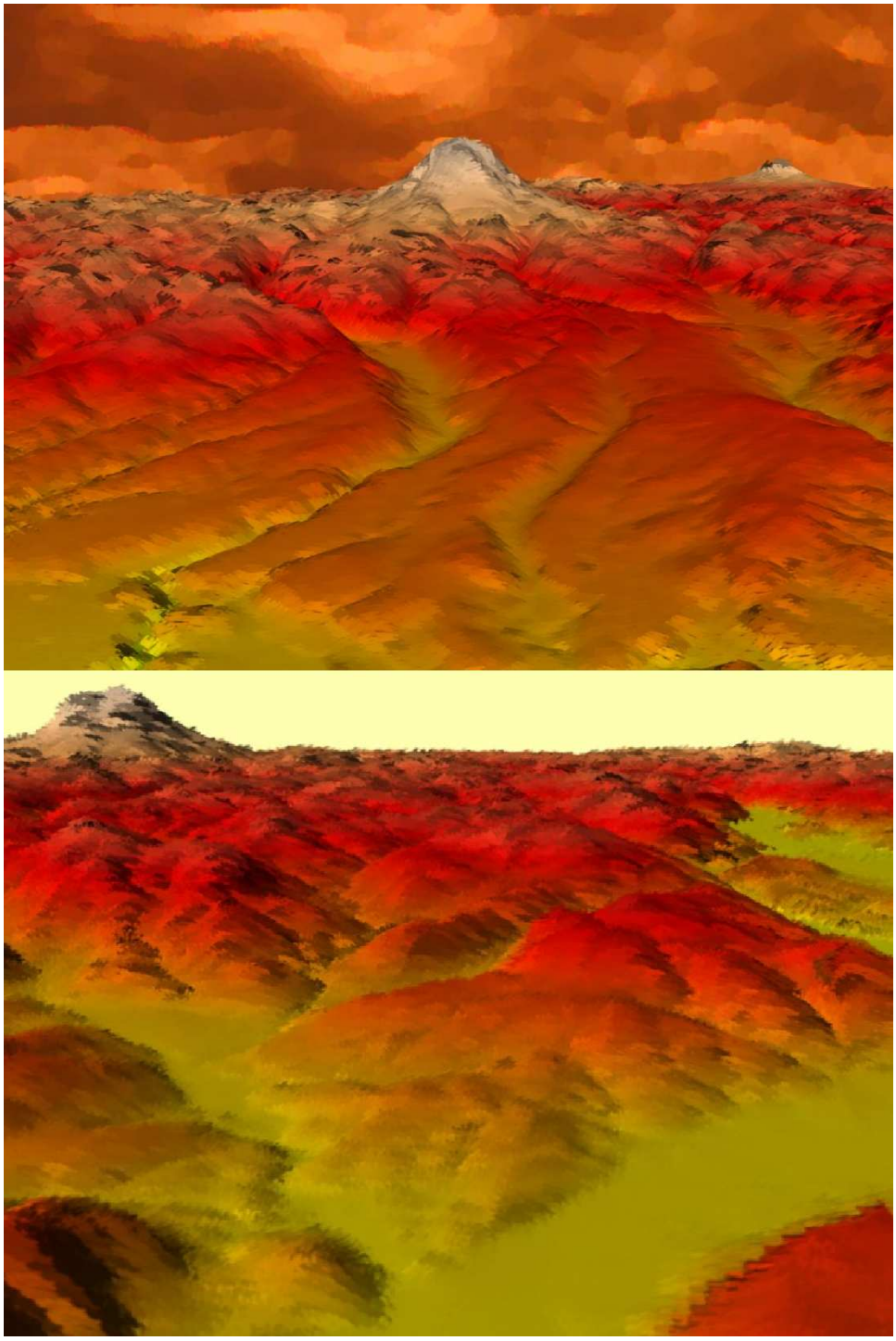

Figure 6.17 Mountains and valleys in Puget Sound painterly rendered.

Chapter 7

Conclusions

In this thesis, we presented methods to render terrains in real-time on commodity GPUs. Our aim was to find methods for different needs of handling terrains. Terrains sometimes need to be just rendered and sometimes they need processing in real-time. Terrain rendering can be for pure entertainment or can be for informational or educational purposes. We wished to explore ideas which suit this wide requirements and build systems satisfying them.

We presented a system for real-time rendering, deformation, editing, and physics computation of large terrains. The representation enables quick rendering and has the ability to manipulate the terrain on-line. We wanted a system which enables application developers direct and simple accessiblity to the terrain data. For example, applications like video games, in most cases, have a terrain renderer with many other components: Networking for multiplayer, physics and animations for realistic behavior, artifical intelligence of NPCs (non playable characters) for replayability value etc. Our system keeps a low profile on the CPU keeping a lot of processing margin for other components in such applications.

Then we presented a Spherical Terrain Rendering algorithm which provides uniform sampling of points over the surface and fast rendering with low memory usage. Traditional systems used to keep discrete visibility of the planet from ground zero and from space. Tricks used distract the viewer when the flythrough needed a switch between the representations. Our unified representation enables us to view the terrain of the planet in real-time from space to ground level. The reason behind our claim, that this is the best method to do spherical terrain rendering, is it combines two independent and best methods (HTM to index and sample spheres, and Clipmaps to render terrains) in a very simple and direct way.

We also presented a method to render terrains with an artistic style for abstract visuals. Through computer graphics, terrains with a pencil, pen or ink style have been tried before but painting remained a cumbersome task. With our back to front ordering method without the need of sorting, it is possible to have many brush strokes in real-time on the screen mixing with each other, just like an artist mixes them on a piece of canvas. We get nice visuals with frame to frame coherence on animation of the scene. Considering rich nature of terrains and cumbersome nature of painterly rendering processes, we get good performance with our system using latest graphics hardware. This system opens a lot of possibilities for

artists as they can experiment with creating different type, or density, or orientation of strokes. Different artistic styles can be achieved with a variety of taste.

We conclude this thesis by mentioning a number of possiblities for future work. Our terrain renderer requires all data to be present at the main memory since our primary focus was creating a GPU Cache based system. A scheme very similar to what is used for the GPU cache can be used to manage the data on the CPU at an appropriate resolution. Since the CPU cache will need occasional updates from the disk, we can update it with a parallel low priority thread using today's dual core or better processors. With introduction of Solid State Drives as a commodity, caching will have even better performances.

In spherical terrain rendering system, other than just rendering, HTM's geospatial indexing capabilities can be used to create a search engine for geological, topological or simple geographical information. Apart from above, our memory usage right now is $9/16$ of what we consume in the memory. Since our system has a low memory footprint in the first place, this doesn't seem to be a problem. Nevertheless fitting usuable terrain data in unusable regions of our clipmaps will be a welcomed improvement.

We introduced painterly rendering of terrains and we believe there is a lot of future work needed. According to emerging painterly styles, they can be algorithmically analyzed for working on terrains along with aesthetical analysis of the results. The idea of rendering procedural or vector strokes (similar to geo-graftals mentioned in [39] and [30]) instead of current bitmap strokes is already thrilling. This will open even more options for artists to create varied visuals.

* * *

Related Publications

Conference Papers:

- Shiben Bhattacharjee and P. J. Narayanan.
  Hexagonal Geometry Clipmaps for Spherical Terrain Rendering,
  Sketch, in *The 1st ACM SIGGRAPH Conference and Exhibition in Asia* (SIGGRAPH Asia), 2008.

- Shiben Bhattacharjee, Suryakant Patidar and P. J. Narayanan.
  Real-time Rendering and Manipulation of Large Terrains,
  Paper, in *Sixth Indian Conference on Computer Vision, Graphics & Image Processing* (ICVGIP), 2008.

- Shiben Bhattacharjee and P. J. Narayanan.
  Real-time Painterly Rendering of Terrains,
  Paper, in *Sixth Indian Conference on Computer Vision, Graphics & Image Processing* (ICVGIP), 2008.

- Soumyajit Deb, P. J. Narayanan and Shiben Bhattacharjee.
  Streaming Terrain Rendering,
  Sketch, in *The 33rd International Conference and Exhibition on Computer Graphics and Interactive Techniques* (SIGGRAPH), 2006.

- Shiben Bhattacharjee, Soumyajit Deb, Suryakant Patidar and P. J. Narayanan.
  Real-time Streaming and Rendering of Terrains,
  Paper, in *Fifth Indian Conference on Computer Vision, Graphics & Image Processing* (ICVGIP), 2006.

- Shiben Bhattacharjee and Neeharika Adabala.
  Texture Guided Realtime Painterly Rendering of Geometric Models,
  Poster, in *Fifth Indian Conference on Computer Vision, Graphics & Image Processing* (ICVGIP), 2006.

Technical Reports:

- Suryakant Patidar, Shiben Bhattacharjee, Jagmohan Singh and P. J. Narayanan.
  Exploiting the Shader Model 4.0 Architecture,
  Technical Report, IIIT Hyderabad, 2006.

Bibliography

[1] Concise bibliography of the history of cartography *(http://www.newberry.org/collections/conbib.html)*, The NewBerry Library.

[2] A. Asirvatham and H. Hoppe. Terrain rendering using gpu-based geometry clipmaps. *GPU Gems 2*, pages 46–53, 2005.

[3] S. Atlan and M. Garland. Interactive multiresolution editing and display of large terrains. *Computer Graphics Forum*, 25(2):211–223, 2006.

[4] S. Bhattacharjee and N. Adabala. Texture guided real-time painterly rendering of geometric models. In *5th Indian Conference, ICVGIP 2006*, pages 311–320. LNCS 4338, 2006.

[5] S. Bhattacharjee, S. Patidar, and P. J. Narayanan. Real-time rendering and manipulation of large terrains. In *6th Indian Conference, ICVGIP*, 2008.

[6] J. Blow. Terrain rendering at high levels of detail. In *Game Developers Conference*, 2000.

[7] D. Blythe. The direct3d 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM Press.

[8] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Bdam - batched dynamic adaptive meshes for high performance terrain visualization. *Comput. Graph. Forum*, 22(3), 2003.

[9] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Planet-sized batched dynamic adaptive meshes (p-bdam). In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 20, Washington, DC, USA, 2003. IEEE Computer Society.

[10] P. Cignoni, E. Puppo, and R. Scopigno. Representation and visualization of terrain surfaces at variable resolution. *The Visual Computer*, 13, 1997.

[11] M. Clasen and H.-C. Hege. Terrain rendering using spherical clipmaps. In *EuroVis 2006: Symposium on Visualization*, pages 91–98, 2006.

[12] L. Coconu, O. Deussen, and H.-C. Hege. Real-time pen-and-ink illustration of landscapes. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 27–35, New York, NY, USA, 2006. ACM Press.

[13] D. Cohen-Or and Y. Levanoni. Temporal continuity of levels of detail in delaunay triangulated terrain. In R. Yagel and G. M. Nielson, editors, *IEEE Visualization '96,*, pages 37–42, 1996.

[14] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.

[15] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Comput. Graph. Forum*, 18(3):83–94, 1999.

[16] L. D. Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In *IEEE Visualization*, pages 103–110, 1997.

[17] R. Geiss. *Generating Complex Procedural Terrains Using the GPU*. Addison Wesley, 2007.

[18] T. Gerstner. Multiresolution compression and visualization of global topographic data. *Geoinformatica*, 7(1):7–32, 2003.

[19] P. Haeberli. Paint by numbers: abstract image representations. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 207–214, New York, NY, USA, 1990. ACM Press.

[20] M. Haller and D. Sperl. Real-time painterly rendering for m.r. applications. In *GRAPHITE '04: Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 30–38, New York, NY, USA, 2004. ACM Press.

[21] J. Hays and I. Essa. Image and video based painterly animation. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, pages 113–120, New York, NY, USA, 2004. ACM Press.

[22] Y. He, J. Cremer, and Y. E. Papelis. Real-time extendible-resolution display of on-line dynamic terrain. In *Graphics Interface*, 2002.

[23] A. Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 453–460, New York, NY, USA, 1998. ACM.

[24] A. Hertzmann. Fast paint texture. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, New York, NY, USA, 2002. ACM Press.

[25] A. Hertzmann. Tutorial: A survey of stroke-based rendering. *IEEE Comput. Graph. Appl.*, 23(4):70–81, 2003.

[26] A. Hertzmann and K. Perlin. Painterly rendering for video and interaction. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 7–12, New York, NY, USA, 2000. ACM Press.

[27] D. Hill and D. Hill. An efficient, hardware-accelerated, level-of-detail rendering technique for large terrains. Technical report, University of Toronto, 2002.

[28] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[29] M. V. J C Whelan. Formulated silhouettes for sketching terrain. In *Proceedings of Theory and Practice of Computer Graphics 2003*, pages 90–97, Birmingham, UK, 2003.

[30] M. Kaplan, B. Gooch, and E. Cohen. Interactive artistic rendering. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 67–74, New York, NY, USA, 2000. ACM Press.

[31] P. Kipfer, M. Segal, and R. Westermann. Uberflow: a gpu-based particle engine. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, page 24, New York, NY, USA, 2004. ACM.

[32] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 123–131, 2004.

[33] J. Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 259–266, Washington, DC, USA, 2002. IEEE Computer Society.

[34] X. Li and J. M. Moshell. Modeling soil: realtime dynamic models for soil slippage and manipulation. In *SIGGRAPH*, pages 361–368, 1993.

[35] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH*, pages 109–118, 1996.

[36] P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[37] Y. Livny, Z. Kogan, and J. El-Sana. Seamless patches for gpu-based terrain rendering. *Journal of WSCG*, 15(1–3), 2007.

[38] F. Losasso and H. Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, 2004.

[39] L. Markosian, B. J. Meier, M. A. Kowalski, L. S. Holden, J. D. Northrup, and J. F. Hughes. Art-based rendering with continuous levels of detail. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 59–66, New York, NY, USA, 2000. ACM Press.

[40] B. J. Meier. Painterly rendering for animation. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484, New York, NY, USA, 1996. ACM.

[41] S. O'Neil. Rendering planetary bodies, Gamasutra, 2001.

[42] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *IEEE Visualization*, pages 19–26, 1998.

[43] S. Patidar, S. Bhattacharjee, J. Singh, and P. J. Narayanan. Technical report on shader model 4.0 architecture. Technical report, IIIT Hyderabad, India, 2007.

[44] A. Santella and D. DeCarlo. Abstracted painterly renderings using eye-tracking data. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, New York, NY, USA, 2002. ACM Press.

[45] J. Schneider, T. Boldte, and R. Westermann. Real-time editing, synthesis, and rendering of infinite landscapes on gpus. In *Proceedings of Vision, Modelling, and Visualization*, 2006.

[46] J. Schneider and R. Westermann. Gpu-friendly high-quality terrain rendering. *Journal of WSCG*, 14(1-3):49–56, 2006.

[47] M. Shiraishi and Y. Yamaguchi. An algorithm for automatic painterly rendering based on local source image approximation. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 53–58, New York, NY, USA, 2000. ACM Press.

[48] A. Szalay, J. Gray, G. Fekete, P. Kunszt, P. Kukol, and A. Thakar. Indexing the sphere with the hierarchical triangular mesh. Technical report, (MSR-TR-2005-123), Microsoft Research, 2005.

[49] D. Wagner. Terrain geomorphing in the vertex shader. *ShaderX2, Shader Programming Tips and Tricks with DirectX 9, Wordware Publishing*, 2004.