REAL-TIME SIMULATION AND VISUALIZATION OF DEFORMATIONS ON HEIGHTFIELDS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BİLKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

M. Adil Yalçın

June, 2010

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Bülent Özgüç (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Tolga Çapın (Co-Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Uğur Güdükbay

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Veysi İşler

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

REAL-TIME SIMULATION AND VISUALIZATION OF DEFORMATIONS ON HEIGHTFIELDS

M. Adil Yalçın
M.S. in Computer Engineering
Supervisor: Prof. Dr. Bülent Özgüç
Co-Supervisor: Asst. Prof. Dr. Tolga Çapın
June, 2010

The applications of computer graphics raise new expectations, such as realistic rendering, real-time dynamic scenes and physically correct simulations. The aim of this thesis is to investigate these problems on the heightfield structure, an extended 2D model that can be processed efficiently by data-parallel architectures. This thesis presents methods for simulation of deformations on heightfield as caused by triangular objects, physical simulation of objects interacting with heightfield and advanced visualization of deformations. The heightfield is stored in two different resolutions to support fast rendering and precise physical simulations as required. The methods are implemented as part of a large-scale heightfield management system, which applies additional level of detail and culling optimizations for the proposed methods and data structures. The solutions provide real-time interaction and recent graphics hardware (GPU) capabilities are utilized to achieve real-time results. All the methods described in this thesis are demonstrated by a sample application and performance characteristics and results are presented to support the conclusions.

Keywords: Computer graphics, computer animation, physical simulation, heightfields, deformations.

ÖZET

YÜKSEKLİK HARİTALARI ÜZERİNDEKİ ŞEKİL DEĞİŞTİRMELERİN GERÇEK-ZAMANLI SİMÜLASYONU VE GÖRSELLEŞTİRİLMESİ

M. Adil Yalçın
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Bülent Özgüç
Tez Yardımcı Yöneticisi: Asst. Prof. Dr. Tolga Çapın
Haziran, 2010

Bilgisayar grafiği uygulamaları gerçekçi görselleştirme, gerçek-zamanlı dinamik ortamlar ve fiziksel olarak modellenmiş nesnelerin simülasyonu gibi gereksinimleri de barındırmaktadır. Bu tezin amacı, söz konusu problemleri yükseklik haritası modeli üzerinde araştırmaktır. Yükseklik haritası, genişletilmiş 2 boyutlu bir modeldir ve bu model, veri-paralel bilgisayar mimarilerinde verimli şekilde işlenebilmektedir. Bu tez, üçgensel objelerin etkisiyle oluşan yükseklik haritaları üzerindeki bozulmaların simulasyonu, objelerin yükseklik haritası üzerinde fiziksel simulasyonu ve şeklen bozulmuş bölgelerin ileri görselleştirilmesi konularında yöntemler sunmaktadır. Hızlı görselleştirme ve daha kararlı fiziksel simulasyon elde etmek amacıyla, yükseklik haritası, iki farklı çözünürlükte saklanmaktadır. Sunulan yöntemler, geniş ölçekli bir yükseklik haritası yönetim sisteminin parçaları olarak gerçeklenmiş olup, ek olarak detay seviyesi ve optimizayson algoritmaları da içermektedir. Sonuçlar, gerçek-zamanlı etkileşime olanak tanımakta ve güncel grafik işleme donanımları (GPU), gerçek-zamanlı sonuçlar alınmasına yardımcı olmaktadır. Bu tezde tanımlanan yöntemler, örnek bir uygulamayla gösterilmekte ve varılan sonuçları doğrulamak üzere performans değerlendirmeleri sunulmaktadır.

Anahtar sözcükler: Bilgisayar Grafiği, Bilgisayar Animasyonu, Fiziksel Simulasyon, Yükseklik Haritası, Geometrik Deformasyon.

Acknowledgement

First and foremost, this thesis would not have been possible without the care and support of my mother and the protection of my father.

I thank Damla Arifoğlu for standing by my side in my good and bad days, her support and love have been invaluable. My friend Mehmet Koçakoğlu has been like a brother to me since I have known him, lending a hand and a mind when I was in need. I also want to thank my friends from Middle East Technical University and Bilkent University (Bahadır, Caner, İlkay and others) with whom I shared valuable times. I am grateful to my friends Sefa and Doğa for letting me use their PC (PC1) for testing the sample application.

I feel privileged to have worked with my advisors through the course of my M.Sc. studies. I thank Bülent Özgüç for making research so fun and exciting. I learned so much from his vision and my conversations with him have always been rewarding. I thank Tolga Çapın for supporting my dreams and always valuing my ideas, whether they may be right or not. He has provided me an environment in which I could develop my skills and thus the methods presented in this thesis. I also want to thank the members of the thesis jury, Uğur Güdükbay and Veysi İşler for evaluating the thesis and providing their feedback. Moreover, they have introduced me various problem domains of computer graphics within their courses, which I loved to attend as a student.

I want to express my sincere thanks to all the people around the world who have contributed to the open global knowledge. This spans not only the contributors of the open source software libraries that have been used in this thesis, but also contributers of all open source libraries and all open and reliable information sources and arts in many forms. And lastly, I want to thank many talented and open minded, independent musicians for making the atmosphere a better place to live in with their unique sound vibrations.

My M.Sc. studies have been financially supported by TÜBİTAK (The Scientific and Technological Research Council of Turkey) B IDEB scholarship and also partly by European Union through 3DPHONE project.

Dedicated to my mother...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Heightfield, also commonly called heightmap or terrain, is one of the basic graphical structures that is commonly used in virtual 3D scenes. Heightfield is a uniform discrete grid-based structure, in which each cell holds a relative height information with respect to a ground height level. Thus, this structure especially fits well into outdoor scenes to provide a basic solid world geometry, such as valleys, mountains, hills, cliffs and smaller perturbations on ground. Other scene objects, such as vegetation, characters and vehicles, are generally placed upon this graphical structure. Heightfields can also be utilized and exploited in texturing techniques. Their uniform grid-based structure further allows solving computationally complex problems more efficiently, including collision detection.

While there exists extensive research on heightfields, mostly in rendering optimizations (such as continuous or discrete level-of-detail systems), shading (including shadowing), procedural generation and erosion models [47, 2], there currently exist no real-time and scalable architecture to support physical interactions between arbitrary 3D objects and heightfield structures. This thesis aims to provide a system which considers collision detection, physical simulation, level-of-detail optimizations, rendering and editing problems as a whole. The proposed system is designed to work efficiently on recent parallel GPU architectures, a hardware dedicated to graphic processing, currently available in most consumer-level hardware. The usage of GPU, with its parallel computing powers, in turn allows significant speed-ups and real-time computation of dynamic heightfield structures.

1.1 Motivation

To generate a realistic virtual environment, it is necessary that the environment dynamically updates itself. The wind should move the leaves of trees, the rain should make the surfaces look wet, jet planes should leave trails behind and a virtual character should leave footsteps behind when walking on the ground. Making a virtual environment highly responsive to object's physical dynamics and interactions in-between different objects and models so that it can better represent the real world phenomena will enhance the viewers experience. Dynamic virtual environments should be able to generate these secondary phenomena to add visual and physical consistency to the scene.

Although the interaction details can be produced as a post-production step if the results are generated offline in non-interactive frame-rates, merging such phenomena in a post-processing stage has its own problems, since interaction between separate computational models are hard to manage and achieving realistic results requires heavy human intervention. Integrating such detail layers into computer graphics applications is only possible through automated algorithmic approaches that can support artistic control as well.

The basic motivation behind this thesis is the absence of a theoretical and practical work focusing on real-time physical interactions with the heightfields, which involves two different geometric models, 2D grid-based heightfields and arbitrary 3D vertex-based meshes. To the best of our knowledge, no real-time application exists that includes routines to deform the terrain as a result of a character walking on it. Yet, also in most of the recent production CG movies, one can observe that objects do not leave trails or marks on the ground.

Since the problem domain is open to data-parallel approaches on heightfield structures, the GPU is used in many phases of the proposed method. With its recent capabilities, such as unified shader architectures with additional pipeline stages and single/double precision data type support, GPU hardware has been used to as a general-purpose parallel processor in many problem domains, including but not limited to, physical simulation, signal and image processing, segmentation, global illumination and database systems [35].

1.2 Challenges

The greatest challenge in this work is the focus on real-time simulations over relatively large scale data, which can contain heightfields of size 4096 x 4096 and hundreds of objects with thousands of polygons. Introducing a level-of-detail system for calculating and visualizing deformations is thus required to achieve visual and simulation quality where it will be more visible.

Since the GPU is used in many parts of the system proposed, scaling the algorithms to GPU architectures so that significant speed-ups can be achieved stands as another challenge. Communication between the CPU and the GPU is a known bottleneck in many applications, thus it is aimed in this work to shift the computation load as much as possible to GPU, which can process large data structures faster than CPUs, using parallel computing units.

Exact physical simulations are computationally hard to solve, and hard to model as well, since the nature is so complex that still many of its rules are unknown. Likewise, more precise physical models may not work real-time, both because of their computational complexities and possibly larger data sizes. As a result, this thesis focuses on appearance-based models that can produce convincing visuals, rather than realistic simulations. The deformations can result in the volume of the material to be changed. The proposed material parameters also do not aim to reflect scientific material properties and are defined to allow ad-hoc modifications.

A requirement to achieve robust simulation is scalability over multiple objects in the virtual scene. There can be many objects in the scene interacting with a heightfield structure, and each of these objects need to be simulated concurrently. In this thesis, it is also assumed that terrain-object interactions cover only a subset region of a large-scale terrain, thus efficient broad-phase collision detection can be applied to decrease active data set sizes for collision, compression and decompression pipeline.

Physical consistency also has to be maintained. As an example, assume a car drives over a snow-filled road. The snow is compressed by the car and the car follows the compressed road(terrain). If another car passes through the same

compressed marks later, the ground is not compressed again and second car can follow the same track as the first one. The car example above implies that managing physical simulation states, which can, for example, result in limited compressibility, is a factor that must be addressed to build a complete system. Also, the compressed regions should also have a dynamic behaviour, letting them converge to a more physically plausible state with respect to initial conditions after deformations occur.

Since the height data is updated dynamically, techniques that exploit static features of data structures using pre-computations to achieve speed-ups may not be used in the proposed system. This puts further restrictions on the techniques that can be applied. If required, pre-computed data needs to allow real-time local updates in such a system.

Rendering of the dynamic structures is a problem heavily linked to the underlying data structures. Rendering enhancements and optimizations can collaborate with the dynamic simulation of deformations on the ground to create more visually appealing and convincing results and to increase the rendering speed without sacrificing quality.

1.3 Overview of the System

The proposed system consists of terrain deformation pipeline, rigid body simulation on terrain and visualization pipeline, using shared static and dynamic data structures in all methods. Figure 1.1 shows the overview of deformation pipeline in the proposed system. The visualization pipeline is composed of multi-resolution block-based level-of-detail and culling methods on low-resolution heightfield data and GPU shading algorithms to render both the deformed and non-deformed heightfield cells. The presented GPU shading algorithms are also analyzed with respect to rendering speed and a level-of-detail system for rendering deformations on ground is presented.

To generate plausible dynamic deformations on heightfield as a result of heightfield-object interactions, the following steps are performed serially in each

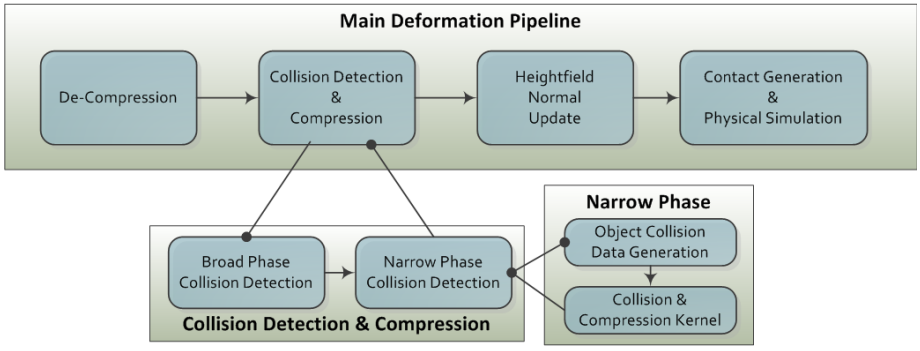


Figure 1.1: The Basic Deformation Pipeline

iteration:

1. De-compression of compressed heightfield cells: Different parameterizable decompression models are developed and discussed in this thesis. This step can be performed efficiently on the GPU.

2. Broad-phase collision detection: This step detects candidate colliding terrain patch - object pairs, operating on the CPU.

3. Narrow-phase collision detection

   (a) Object collision data generation: This step generates required collidable object data for the next step on the GPU.

   (b) Exact collision detection: Using object and terrain collision data, per-cell collision and compressions are applied, using a kernel program that runs on the GPU.

4. Reading back collision results to the CPU and generating contact information between objects and the heightfield

5. Simulating scene objects on the ground using a physical simulation engine. The simulation is performed on the CPU.

In this process, many parts can be parametrized for different material behaviours and artistic control. Parametrization can be per-height-cell, per-patch, global or per-object, in which physical properties of the objects, such as mass,

can be taken into account. The parameter customizations enable the environment designer to represent terrain materials that behave differently at collision time.

One limitation of the implementation of the proposed system is that the objects interacting with the terrain are assumed to be non-deformable. To implement deformable objects, one would have to solve a system of equations which take object deformation properties into account as well. The non-deformable object assumption is chosen to simplify the simulations in order to achieve real-time iterations and to increase stability of the system. Yet, an initial approach to deformable object simulation can use the same heightfield collision contact data on deformable scene bodies.

Another limitation arises from the fundamental problem with the heightfield data structure: Given a point on the ground plane, the heightfield can define only a single height, thus you cannot model hanging cliffs or caves using this data structure.

1.4 Summary of Contributions

The main advantage of the proposed system over texturing-based methods, such as using projected decals, is that the underlying geometry of the heightfield to be rendered is updated as a result of terrain-object interactions. The proposed system also solves efficient object - heightfield collision responses using the deformation data. Texture decals cannot simulate geometric deformations and support further advanced physical simulations, such as decompression. Geometric updates can enhance the visual complexity of the environment to a larger extent than texture-based methods and high resolution contact point generation between heightfield and objects allows realistic intuitive rigid body simulations.

The main contributions of this thesis can be listed as:

- A practical terrain modeling and simulation system incorporating collision

detection, deformation and advanced rendering methods that runs in real-time using CPU and GPU hardware, supporting physical simulation of terrain to object interactions in the virtual scene.

- A two-resolution heightfield data management to speed up simulation and visualization of deformations.
- A culling system that speeds up heightfield-object intersection tests.
- A visualization pipeline that can blend deformed and non-deformed regions using advanced per-vertex and per-pixel shading techniques, and a level-of-detail system for rendering deformed regions.

1.5 Structure of the Thesis

In this thesis, the related work in the problems stated above will be discussed in the next chapter. The basic data structures used in the proposed collision and rendering pipeline are presented next. The discussions then focus on collision detection, compression, decompression and erosion, to identify physical simulations on the heightfield structure. The second phase of the general simulation framework, simulations of objects that collide with ground, is provided in Chapter 5. The description of proposed methods is completed with focusing on the rendering pipeline and a final discussion on the level-of-detail management from rendering perspective. This thesis is concluded with further implementation details, performance measurements, discussions and future work ideas.

Chapter 2

Background and Related Work

This chapter focuses on major related studies and approaches in the problem domains of this thesis. After heightfield specific methods are presented, some of the techniques that have been developed to support generic collision detection algorithms on the GPU are summarized.

2.1 Methods for Heightfield Structures

This section lists some of the key studies that utilize heightfield structures directly.

2.1.1 Deformations

One of the first research results on interactions between terrain and objects was achieved by Li and Moshell [26]. Their method models soil and object interaction, incorporating Newtonian physics and analytical models. The volume conservation constraints and the physical properties of the soil, such as shear stress and shear strength, are analyzed and digging, cutting, piling, carrying and dumping interactions have been studied. The proposed soil structure has two separate models, one is a heightfield, and the other is stored as discrete chunks to model places where soil is pushed by a blade (or some other object). Their approach

follows physically based, rather than appearance based, approaches. Yet, since realistic soil dynamics is hard to model, their method discounts some of the factors involved.

Sumner et al. [43] propose another physical deformation algorithm on a heightfield-based terrain. The algorithm consists of three basic steps, collision detection, displacement and erosion, and an enhancement step, particle generation. The collision detection in this work uses ray-casting from each heightfield cell to candidate intersecting objects. The penetration amount is calculated and the ground is compressed. A distance contour map is generated from colliding cells. This distance map is used in the displacement step. After material displacement, the contours of colliding region store excessive material, which is then eroded into non-colliding regions iteratively in later time-steps of the simulation. In this work, various properties of the compression and decompression algorithm can be controlled by terrain material properties, including liquidity, roughness, inside slope, outside slope and compressibility.

Onoue and Nishita [34] extend the work of Sumner et al [43]. Their method follows the same basic steps, and additionally, it can represent granular materials on top of objects. In the proposed algorithm, objects and granular material on them are represented by two-dimensional array of height spans (Height Span Map, or HSM), while the material on ground surface is still represented with a heightfield structure. Another extension provided by Onoue and Nishite [25] allows collision detection step to be performed on the GPU. This method utilizes depth and stencil buffers to detect colliding regions.

Aquilo et al. [3] describe another method that deals with terrain-object interactions and utilizes the GPU hardware. The difference of this method is that it can deal with terrain compressions using a data structure called Dynamically-Displaced Height Map (DDHM) stored on the GPU. The paper does not discuss displacement and erosion operations and only performs compression.

Yefei He [21] focuses on the off-round vehicle simulation and is based on a dynamic interactive, deformable terrain. This study includes a multi-resolution system, called DEXTER (Dynamic Extension of Resolution), and the visualization follows an extended ROAM algorithm [13].

Zhu and Bridson [48] approach sand animation problem as a fluid-simulation problem. Their sand model is a particle-based model, without the use of height-field structures. To achieve a sand simulation model, they introduce inter-grain and boundary friction into their water simulation model.

Another method that deals with modeling of soil is described by Chanclou et al. [7]. The ground is modeled as an elastic sheet and the interactions are based on particle interactions between the ground and the object. The granular ground is composed of point masses linked by thresholded viscoelastic collision interactions. The ground sheet deforms when the objects interact with the terrain.

2.1.2 Erosion

One of the approaches to simulate terrains is to apply dynamic erosion, which is the result of physical properties of terrain materials and possibly of continuous water, temperature or wind effects. Musgrave et al. [33] present both a locally controllable fractal terrain generator and an erosion model that aims to simulate hydraulic and thermal effects. The hydraulic erosion model takes terrain material transfer from upper to lower regions into account. Thermal weathering is modeled as a simple slope-based restriction over terrain grid cells. Anh et al. [2] present a simple procedural heightfield erosion algorithm that can be executed in GPU hardware. Their method uses multiple channels in heightfield textures, stored on GPU, to store additional data (water amount, dissolved sediment and 3D velocity of water). Benes and Forsbach [4] propose using multiple layers of data on a basic 2D heightfield structure, where each layer element can hold multiple data representing physical properties within that layer, including density, gaseousness and saturation coefficients. They later show an application of thermal erosion algorithm on their data structure. Štava et al. [47] further extend the GPU-based hydraulic erosion implementation to support interactive terrain modeling. Their system combines multiple hydraulic erosion models, uses the multi-layer heightfield representation ([4]).

2.1.3 Level Of Detail

Terrain rendering, and especially level-of-detail systems, have been studied for decades, and they are still open to further advancements, such as development of fast GPU ray-casting methods [11]. Most of the recent popular practical approaches for implementing level-of-detail management into terrain rendering are discussed in [31]. In this work, quad-tree and binary triangle trees are presented as common hierarchies for multi-resolution storage. Basic generic approaches for avoiding cracks / t-junctions in different LOD layers of neighboring triangles are presented. Some of the further discussions include texture-mapping issues and terrain data paging techniques, including using operating system paging API's, tiled pyramids and support for networked streams of data. Multi-resolution terrain models that exploit semi-regularity are also surveyed by Pajarola and Gobbetti [36]. The regular / semi-regular connectivity in this work denotes that the vertices are well-ordered, as in common 2D heightfields structures.

Continuous LOD systems update the triangulation of a surface with triangle insertions/removals based on some error metrics that generally depend on the visible screen-space difference after the update. Lindstrom et al. [28] generate new LOD layers through bottom-up refinement of terrain geometry and use vertex dependencies to prevent cracks when vertices are removed. Efficient triangle strip indexing for vertices to be rendered is generated by a top-down recursive traversal. Another popular continuous algorithm, ROAM [13] (Realtime optimally adapting meshes), is based on a binary triangle tree structure and uses priority queues to track the vertices to be removed or inserted as required. As another continuous LOD approach, Hoppe adapted his View Dependent Progressive Meshes to terrain structures [22]. The triangulations in this work are more similar to TIN-based models than grid-based models, providing better approximations with a given number of vertices. Geometry clipmaps [30] is a paged (off-the-core) algorithm that can visualize very large datasets. It is based on a nested regular grid structure around the viewpoint, which can be incrementally updated as the camera moves.

2.1.4 Surface Details with Heightfields

Another use of heightfield data structures appears in shading and texturing of 3D models. This approach commonly involves associating a heightfield map to the surface of a 3D polygonal model. Using heightfields as a displacement factor was proposed by Cook [9], and its implementation generally follows per-vertex calculations, offsetting 3D coordinates of input vertex given a heightmap as a displacement map. On the other hand, per-pixel image-based rendering techniques, as will be summarized in this section, calculates per-pixel displacement vectors for points during rendering. They commonly use an inverse displacement mapping method, trying to find the position on the model which is to be seen on the output. The heightmap is often scaled to $[0, 1]$ range to ease the creation of art assets.

The first study that discusses this approach is parallax mapping [24]. This technique extends bump mapping technique [5], which introduces perturbations to pixel normals. A parallax factor, representing the displacement offset on the surface is calculated. To find this offset, the height of the rendered fragment on the surface is sampled by the pixel shader and the input texture coordinate is moved along the view direction using this parallax offset. This method can provide a very fast approximation to parallax effect in the surface since it only involves a single additional lookup and simple offset calculation mathematics. Yet, this method suffers from computational errors when the heightfield is sharp, when there are heightfield occlusions along the view direction or when the viewer observes the surface at oblique angles.

Recent extensions of the basic approach that aim to shift texture coordinates using parallax offset employ more precise heightfield intersection routines and so can better render self occlusions and avoid artifacts that cause texture floating on surface. In the methods described below, intersections are calculated mostly in 3D texture-space (u-v-t) of a triangle, to be able to automatically parallax map arbitrary polygonal u-v mapped models. This implies a conversion from model to texture space when required, using surface tangent-bitangent-normal (TBN) matrix.

Steep parallax mapping [32] uses a linear search along the view ray to find the

intersection position. They also apply parallax mapping for fur rendering, which is modeled as a height map with peaks (pins/hairs). Using mip-mapping LOD bias for filtering, they claim to find the ray intersections more precisely when dealing with such high frequency features in heightfields.

Policarpo et al. [39] propose a relief texture mapping technique in texture space. They show that using only binary search along the view ray can generate incorrect intersection points in cases where there are multiple heightfield intersections along the ray. Thus, they start with a linear search to find the first intersection along the ray, and refine the result using binary search to find the exact intersection point. This work also introduces dual-depth relief textures, which extends the heightfield image to include inverse heightmap for the back of the model and allows capturing the surface of the back of the object when the object is viewed from oblique angles.

Risser et al. [40] proposes an acceleration to relief mapping method, called interval mapping. Their method is similar to secant root finding method and can converge faster to the ray-heightfield intersection point since it can generate better approximations to final point along the path. In their proposed method, they first start with a regular linear search to find initial upper and lower intersection bounds. Then, instead of selecting the mid point between the bounds as a new bound for the binary search, they select the intersection of the view ray and ray between bounding heights.

Tatarchuk [45] proposes using an adaptive linear sampling distance that depends on the view angle, and a high-precision fast approximation of the intersection point after the linear intersection routine. Her work also focuses on generating efficient soft self-shadowing and introducing an adaptive level-of-detail system, which falls back to simpler bump mapping rendering technique. This level-of-detail scheme uses the computed mip-map level in the pixel shader as the level-of-detail metric.

There are also extensions on methods for ray marching. Donnelly et al. [12] extend the heightfield-ray intersection approach using a pre-computed 3D distance map texture, which stores the distance to the closest point on the surface to be viewed. Their ray marching technique is based on sphere tracing [20], a

technique developed to accelerate ray tracing of implicit surfaces. They use the 3D distance map to choose dynamic sampling intervals along the ray, which is computed at each step. This allows fast convergence to the real intersection point, while not skipping any intersection because of under sampling issues, as can be the case with linear search.

Cone Step Mapping (CSM) [14] reduces the 3D voxel space distance transform proposed by Donnelly et al. [12] to a 2D cone map, reducing the memory requirements significantly. This technique uses a pre-processed 2D texture which stores the maximum angle of a cone, pointing upwards and not touching the heightfield, for each texel of the base heightfield. This cone-map data is used to adjust iteration steps. The resulting algorithm does not miss first intersections with the heightfield since it prevents under-sampling, but may not be able to converge to an intersection if the number of steps is low. Policarpo [38] extends CSM so that it can converge to a result in smaller number of steps. Their method generates larger radius cones, detects a first intersection earlier and then uses precise binary search along the ray to find the correct intersection point. Effectively, they are replacing the linear search, which is prone to sampling errors with a more precise and adaptive relaxed, pre-computed cone step map.

The comparisons and common ideas of the methods described above are:

- When linear search is used, the branches and the points sampled along the view ray are more predictable. However, under-sampling problems can be observed. Firstly, exact intersection point may not be found when step size is inadequate, yet smaller step sizes increase number of samples along the ray and decrease efficiency of the algorithm. This problem is often dealt by using iterative refinement after initial bounds are found. Secondly, linear search based approach can also miss intersecting regions between two consequent sample points in which no intersection occurs, which stands as a more fundamental problem that may require expensive extensions, such as the one proposed in [12].

- Texture sample points are generally adjacent or close, thus the ray intersection tracing algorithms can take performance advantage of texel caching on the GPU. Linear searching is more likely to sample adjacent textures,

thus can result in higher performance even though the number of samples along the ray that are needed to find an intersecting point may increase.

2.2 Collision Detection and Physical Simulation on GPU

Fast and robust collision detection stands as a challenge in simulating virtual environments. The graphical models have no knowledge of the complete scene, thus animations can violate physical consistency. Efficient and robust methods are required to accurately simulate physical environments.

To the best to my knowledge, [23] stands as the most recent and extensive survey categorizing existing approaches and discusses most of the available 3D collision detection methods for non-parallel implementations. Generic spatio-temporal intersections, swept-volume interference, multiple interference detections with adaptive time-steps and trajectory parameterizations are presented as basic collision detection approaches that can process timing information. Later, static interference detection approaches are analyzed under convex and non-convex polyhedra. Time-based and spatial bounding strategies are also categorized. Of higher importance to the methods presented in this thesis, hierarchical bounding volume structures include spatial partitioning representations (octrees, BSP-trees and regular grids) and object partitioning representations (which calculate bounding volumes for object primitives and creates a scene hierarchy). Characteristics of oriented and axis-aligned bounding boxes and related spatial management are presented. It should be stressed that the requirements for generation of tight-fit bounding boxes, fast intersection tests and fast updates of bounding data in dynamic scenes are generally not coinciding. Another survey that focuses on types of geometric models rather than collision detection approaches in general is presented by Lin and Gottshalk [27].

With the recent advances in the of the GPU as a parallel processor, collision detection algorithms are emerging. On GPU, potentially colliding sets (PCSs) can be computed for collision detection purposes. In CULLIDE, Govindaraju

et al. [19] compute a PCS for object level and another refined PCS for sub-object level, followed by an exact triangle collision on CPU. They later extend CULLIDE to detect self (intra-object) collisions and also introduce a new, refined culling algorithm and approach which aims to generate collision free sets [18] and process fewer pairs of objects.

Physical simulation is one of the tasks that can be carried by the recent programmable GPU architectures [35]. Fluid dynamics using Navier-Stokes equations, cellular automata and spring-mass dynamics using partial differential equations are some techniques related to physical simulation that can take advantage of the parallel processing.

Galoppo et al. [16, 17] present a method for simulation of dynamic deformable models using dynamic deformation textures and the internal steps of their method are optimized for parallel hardware (GPU). The deformable surfaces are modeled as a 3D core covered by a deformable layer stored as a 2D texture. They use a two-stage collision detection algorithm, first identifying planar contact regions between deformable models and then executing high-resolution collision and intersection detection. They decouple surface displacement map updates (many DOF's) from the un-deformable core updates (6 DOF's). Their collision response method uses Lagrange multipliers and approximate implicit integration.

Chapter 3

Data Structures

This section presents all the data structures used by the proposed methods in this thesis. It does not describe how this data is used in the methods, but focuses on descriptions and properties of the structures.

3.1 Heightfield Data Basics

A heightfield is a 2D 1-channel uniform grid-based data that stores the sampled height values over a 3D surface given 2D coordinates over a grid on a plane. Figure 3.1 shows a perspective 3D heightfield block rendering. The point samples on the heightfield denote the sampled height values on respective grid corners. In this thesis, the heightfield topology is used to denote both the heights of sampled points and the normals on a triangulation of heightfield surface.

In this thesis, the heightfield data is managed in two separate resolutions which serve distinct set of functions. The low resolution heightfield data is used as the basic terrain topology to be rendered. The high resolution heightfield data is used by collision detection and deformation methods for heightfields, and it can be procedurally generated from low resolution data on demand (i.e. when an object is about to intersect the terrain patch). The proposed visualization (rendering) methods for the heightfield can scale to both low and high resolution heightfield data and aim to render high quality images with seamless deformations.

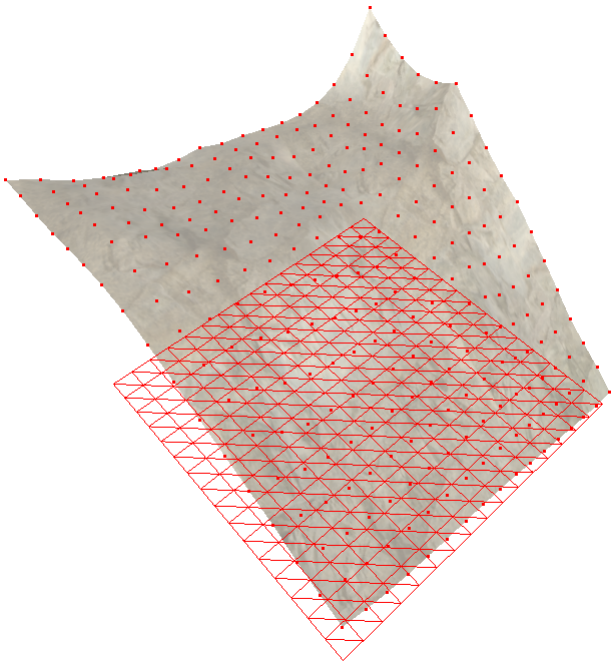


Figure 3.1: A heightfield point and wireframe rendering on its 2D grid base

The low-resolution heightfield data is stored both on the CPU as a terrain attribute image (Section 3.2.4), and on the GPU as a list of 2D textures (Section 3.3.1). CPU-side heightfield data is mainly used for ray-casting during heightfield selection with an input device and also to be able to sample height values given a 2D world-space coordinate on the heightfield grid in the CPU. The high-resolution heightfield data is stored on the GPU only and it is used by the proposed visualization and deformation methods on heightfields.

The initial low-resolution heightfield values are either loaded from external file sources or generated procedurally using Perlin noise algorithms as described in Section 7.1.1. This initial data on CPU is then uploaded to low-resolution GPU textures. Generation of high-resolution heightfield texture data follows triangular interpolation over low-resolution data, identical to the interpolation performed by the GPU given triangle indices. This step aims to reproduce the exact interpolated height values between sample vertices on rendered heightfield data and to avoid height pop-up effects that may appear when switching between high and low resolution heightfields. Thus, the main topology of the heightfield follows low resolution data and this topology is re-sampled to generate high resolution/high

frequency heightfields when required for use in deformation algorithms. In Figure 3.2, notice that low resolution cell boundaries (interpolated by the GPU) with high-resolution cells (interpolated by the CPU) are seamless with respect to vertex height values.

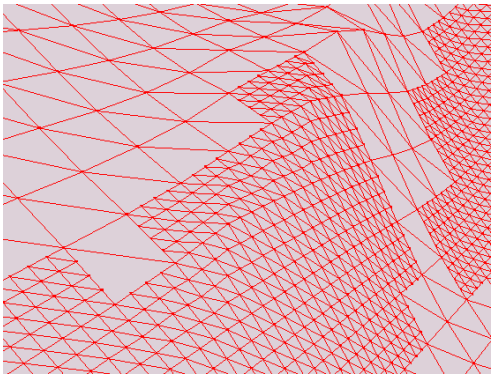


Figure 3.2: Visualization of mixed low-resolution and high-resolution data (as created by triangular interpolation).

In the implementation of this thesis, heightfield per-cell data is selected as 16 bit unsigned integer type. 8 bit data allows only 256 levels for terrain height and fails to represent realistic smooth variations. Using 16 bit data, the heightfields used in this thesis can have a range of 65536 values. This choice has been made to prevent floating point calculation errors that may cause data synchronization and stability problems during deformations and physical simulation of objects contacting the heightfield. In the proposed system, stability of the system is given a higher priority than the realistic simulation of the deformations, thus integer types are preferred. Using 16 bit data per-height-sample is also a memory efficient uncompressed presentation, rather than 32 bit integer or floating point data. Another advantage of using integer data appears on rendering phase. Floating point type data may result in z-buffer fighting when two vertices with the same uploaded coordinates are rendered because of floating point operations precision and approximations on the GPU. With integer data types, the result of arithmetic operations are precise, unless overflows or underflows occur. However, the integer data type choice introduces its own problems, which can be listed as the following:

- The height values cannot be linearly filtered, they can only be sampled using nearest filtering which returns samples at discrete positions over heightfield.

- The deformation simulation cannot generate or use high-precision floating point data. Although the integer data can be converted to world-space floating point data, the conversion is done through a multiplication with a constant floating point value, thus the values that can be generated are limited.

The hardware linear filtering restriction can be solved by implementing required interpolation methods in shaders, although they may not deliver the same performance as an optimized hardware implementation. Also, the vertical resolution of heightfield (16 bit in proposed implementation) is sufficient to support stable physical simulation through contact points. For simulation of deformations on heightfields, the vertical resolution can be further increased using the method proposed below.

The heightfield deformation data can be stored as absolute height values or differential values with respect to the initial high-resolution undeformed data. In this thesis, the deformation data is stored as differential to the undeformed heightfield. If the differential approach is followed, the absolute deformed height needs to be calculated using samples from two textures. Yet, differential storage can allow the following optimizations to be implemented:

- The number of bits required to present deformation information can be reduced. This follows the fact that the terrain deformation size is limited and deformed heights are close to initial non-deformed heights. When the number of bits required to present deformation state is reduced, remaining bits can be used to store other per-height-cell state data or the deformation texture can be compressed using a smaller data type or by packing values from multiple cells into fewer number of texels on the texture.

- The heightfield deformations can be stored in a higher vertical resolution. This approach also exploits the reduction in the number of bits, yet uses the spare bits to increase the vertical resolution. Since the methods work on integer data, an increased vertical resolution allows higher precision computations in the same vertical range.

3.2 Data Managed on CPU

The CPU-side data structures are terrain patches and the associated quad-tree structure, terrain sub-patches, index buffer manager and terrain attribute images. Furthermore, the collision objects are implemented as a part of physics abstraction layer Section 5.1.

3.2.1 Terrain Patch and Quad-tree Structure

Terrain patches are square sub-sections of the heightfield that allows division of large-scale heightfields into small-scale heightfields. All terrain patches capture constant size regions over a whole terrain and when merged together by matching their edges, they form the complete gap-less large scale heightfield. This division of a large-scale region allows efficient culling, local regional editing and level-of-detail optimizations over larger scale heightfields and further allows extending the texture size limits of GPU hardware.

The patches are stored in the leaf nodes of a complete quad-tree. Each internal node of the quad tree holds 4 child quad-tree nodes. The quad-tree is stored as an array of patches, indexed like a quad-heap, avoiding memory fragmentation and allowing fast access to child nodes and parent node. Assuming that the size of the low-resolution heightmap data is $(2^{terrainSize} + 1) \times (2^{terrainSize} + 1)$, generation of terrain patches is done through decomposition of low-resolution heightmap into separate square blocks, where each is of size $(2^{patchSize} + 1) \times (2^{patchSize} + 1)$. It can be seen that the number of patches that is generated is $2^{(terrainSize-patchSize)+1}$ and $terrainSize \geq patchSize$. The height of the quad-tree is then $terrainSize - patchSize + 1$. For example, a typical configuration is as the following: $terrainSize = 10$, terrain grid size $= 1025 \times 1025$, $patchSize = 5$, patch grid size $= 33 \times 33$, the number of patches $= 2^{(10-5)+1} = 64$, the height of the quad-tree $= 10 - 5 = 5$.

Axis-aligned bounding box (AABB) hierarchies are commonly used data structures in many computer graphics algorithms and spatial searching problems [41]. A quad-tree AABB hierarchy can be used to speed up collision and intersection

tests, since intersection tests involving AABB's can be done very fast (6 comparisons of floating point values) and many pairs of collision or intersection tests can be avoided. Each quad tree node stores an AABB, holding the limits of the world-space geometry of the patches under that node. The AABB of a leaf node is computed using low-resolution heightfield data on CPU. Terrain quad-tree is then traversed in bottom-up fashion to generate a bounding-box hierarchy by calculating the bounding boxes for internal nodes as the tight fitting AABB of child bounding volumes. Figure A.5 shows a 3D quad-tree AABB hierarchy constructed from a sample heightfield.

Since quad-tree is based on low-resolution heightfield and low-resolution heightfield is not deformable, the AABB's of patches and internal quad-tree nodes are not updated in the simulation. Although the deformations on high resolution heightfields can change the bounding coordinates along the height axis, since the height differences are expected to be small, the AABB's are not updated. If large deformations on heightfields are expected, the AABB's of patches and internal nodes can be updated from high resolution data as required.

Terrain patches also store regional heightfield geometry data, level-of-detail parameters for low-resolution terrain rendering (see Section 6.1.1) and sub-patches owned (see Section 3.2.2). The geometry data owned by each patch is the vertex displacement maps, as described in Section 3.3.1, and normal maps, as described in Section 3.3.2. Both types of geometric data, in high or low resolution form, are managed on the GPU and the related details are described in Section 3.3.1 and Section 3.3.2.

3.2.2 Terrain Sub-Patch

Terrain sub-patches are structures used by the two-step deformation renderer, as described in Section 6.2.3, to fill in deformed regions over terrain using high-resolution terrain data. This data is not used or needs to be maintained when adaptive rendering methods, as described in Section 6.2.4 and, are active.

Sub-patches correspond to higher-resolution uniform grid squares height information. Each terrain sub-patch region corresponds to a single cell in a low resolution terrain data. Let the size of a sub-patch be $(2^{subpatchSize}+1)\times(2^{subpatchSize}+1)$. Given that the size of a terrain patch low resolution height data is $(2^{patchSize}+1)\times(2^{patchSize}+1)$, the high-resolution height texture size for that patch becomes $(2^{patchSize*subpatchSize}+1)\times(2^{patchSize*subpatchSize}+1)$. Figure 3.3 shows a sample terrain configuration with patch and sub-patch data. In this figure, low-resolution patch data is of size 9×9, sub-patch size is 5×5 and the high-resolution texture size for that patch is 33×33. This figure shows a total of 6 sub-grid data for that grid.

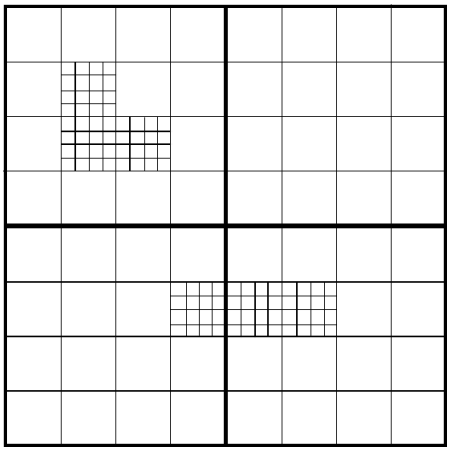


Figure 3.3: A sample small terrain configuration, including 6 sub-patches

Terrain sub-patches are owned and managed by terrain patches. A low-resolution cell in a terrain patch can only have a single or no sub-patch assigned. Initially a terrain patch is not deformed and does not store any sub-patch. The sub-patches are constructed when an object collides and deforms a high resolution terrain cell.

3.2.3 Index Buffer Management

The terrain patches use index data, stored on GPU buffers, to render filled triangles from sampled 3D height vertices. This index data is generated by the

application using an index buffer manager. Given a patch indexing configuration, the manager returns the index data suitable for rendering that patch. The patch configuration is used to transfer rendering low-resolution level-of-detail settings for a patch and includes the self and neighbor LOD levels, and the list of sub-patch data for a patch, if any. The details of the patch index buffer generation method with respect to low-resolution level-of-detail system is described in Section 6.1.2.

Patches having the same patch indexing configuration can share the same index buffer data. For this purpose, the index buffer manager creates index buffers on demand and reuses available buffers when a request with the same configuration is made. This manager also releases index buffers belonging to configurations that are not used by any patch for a configurable number of frames of the application. The configuration used by a patch can be updated each frame since LOD levels are dynamic with respect to camera distance, so index buffer bindings are dynamic as well. In the implementation level, separating the index manager to a separate module helps to achieve a simpler and more effective API.

3.2.4 Terrain Attribute Images

Terrain attribute image data structure is developed to store heightfield parameters directly accessible by the CPU programs. This structure can have varying resolutions and allows interpolating and/or retrieving heightfield parameter values. The supported interpolation methods include bilinear filtering and geometric filtering, which aims to interpolate based on a triangulation over the attribute image. The immediate use of this structure is to store low-resolution heightfield values on the GPU. The implementation supports loading and saving attribute images from/to external files and procedural generation of attribute images as described in Section 7.1.1.

3.3 Data Managed on GPU

The GPU-side data structures are vertex displacement textures, heightfield normal textures, vertex-index buffers and collision buffers. Displacement maps store the height value per-terrain-cell, normal textures store the surface normals over heightfield, vertex-index buffers store the 2D planar shape of a terrain patch/subpatch, and collision buffers are data structures used by the heightfield deformation pipeline of the proposed system.

3.3.1 Heightfield Vertex Displacement Maps

With the introduction of Shader Model 3 (SM3) in Direct3D architecture, vertex shading units are able to sample GPU textures. The hardware support for this operation is also available through recent OpenGL specifications. Additionally, GPU textures can be updated using the graphics rendering pipeline by attaching them as render targets, thus it is possible to dynamically update texture data using the rendering pipeline and to read the updated data in the same pipeline in different stages. These features stand as a key point in the methods developed in this thesis. Texture-based approach to heightfields allows implementing deformation, distribution, erosion and other extensions as image processing operators, which will be discussed in more detail in Chapter 4.

To store vertex height information, 2D GPU textures are used as vertex displacement maps (VDMs) by the vertex shader to set terrain vertex height coordinate given the 2D coordinates of the vertex on a regular 2D grid structure. As discussed in Section 3.2.1 each patch stores its own vertex displacement textures, distributing the heightfield data over multiple GPU textures owned by the terrain patches in the system. Both high- and low- resolution textures are assigned per terrain patch and the textures hold samples of height values over the same region in the virtual world.

One of the vertex displacement textures is a low resolution texture, VDM_{Low}, which stores height/displacement information in low resolution. VDM_{Low} is used to render the parts of terrain which are not physically deformed. This texture is

updated by the CPU as a result of initial heightfield loading or possibly through user editing operations. GPU does not update this texture in any parts of the proposed simulation and rendering pipeline.

The second texture is a higher resolution texture, VDM_{High_NonDef}, which is generated from VDM_{Low} and shares the same height topology. VDM_{High_NonDef} is used to store the non-deformed height values and can be referenced in the shading units to get the non-deformed terrain height. The resolution of this texture is equal to the resolution of terrain patch times terrain sub-patch edge size.

The deformed terrain is also stored as a per-patch data. Another high-resolution texture, VDM_{High_Def}, is used to store this deformed terrain data. The deformation data is stored as a relative data to the undeformed heightfield,VDM_{High_NonDef}. If a grid cell is not deformed, the relative height change is 0. Since VDM_{High_Def} is used as both source and destination texture in many steps in collision detection and deformation pipeline, it actually consists of two separate textures with the same size and type information. This structure is generally known as ping-pong textures. The usage types (source-target) of these two textures are swapped after each GPU program pass that updates the contents of the target VDM_{High_Def}. To sum up, the total number of texture data assigned per patch is 4, and includes VDM_{Low}, VDM_{High_NonDef} and two VDM_{High_Def}.

Initially, patches have no high-res images. High-res textures are created only when an object is about to collide, thus memory usage only increases when required by the simulation. High-resolution textures can be deleted later when they are no more required, freeing up the hardware resources. Although not implemented as a part of this thesis, it is possible to update the low-resolution heightfield from high-resolution compressed heightfield once the high-resolution data is stable and to free high-resolution heightfield memory afterwards. This presents a practical tradeoff between deformed data resolution and memory usage.

3.3.2 Storage of Heightfield Normals

To be able to achieve realistic rendering, surface normals have to be maintained separately for both low and high resolution heightfield data. Since high resolution deformable heightfield data is dynamic, the high resolution normal data needs to be dynamically updated from heightfield. Likewise, the low resolution normal data needs to be updated after initial terrain loading step and after editing operations are applied on low resolution data. Another important factor that needs special care when dealing with normal data in the proposed system is that high and low resolution heightfield rendering need to be blended seamlessly at render time. Surface normals are one of the key components of lighting calculations over surfaces and since parts of terrain will use low-resolution heightmaps and per-vertex lighting calculations while other parts may take advantage of detailed higher-resolution heightmaps and per-fragment lighting calculations, change in normal values used in shading can result in cracks and pop-up effects in real-time renderings.

Per-heightfield-sample normals for 3D meshes can be stored in the two different formats, showing different interpolation and sampling characteristics: as per-vertex attributes or as components of a texture mapped on a surface. Another choice needs to be made regarding when to calculate normal values. Normal values can be generated from heightfield data each time when required, or a separate pass can be applied to generate a normal texture that can be re-used.

Per-vertex normals are sent along with vertex position and other per-vertex attributes, such as color, and this data is only directly accessible in vertex shaders. After being output from vertex shaders, fragment shaders can access component-wise interpolated normals on a surface of triangle generated by three vertices. The interpolation step can produce smooth variations of normal data on the surface of a triangle, but it is not programmable.

Normal textures can be sampled in vertex or pixel shaders. The sample coordinate for normal texture can be a part of mesh data, and the coordinates can be modified in shaders as well. Texture samplers can apply linear filtering to generate interpolated values.

The size of normal data, may it be transfered via per-vertex attributes or as a GPU texture, can be reduced to two components of 3D space. Given two components of a unit normal vector (x and z, for example), one can derive the direction of the remaining component, using the formula $v_y = \pm\sqrt{(v_x)^2 + (v_z)^2}$. If x-z components are mapped to heightfield base plane and y component to heightfield up axis, the normal can only point upwards, limiting the direction of the third component (v_y in the formula above). Thus, it is possible to send only two components of the normal vector and compute the third component in the shaders by using the formula above. If a data type that only supports normalized floating point range [0,1] is used to transfer data, such as GL_FLOAT type in OpenGL specifications, one has to convert normalized [-1,+1] range to [0,+1] range before storing normal vector values and convert [0,+1] range back before generating the third component of the vector.

The proposed system uses GPU textures to transfer heightfield surface normals. The normal textures are owned by terrain patches in both low resolution and high resolution sizes. While generation of normal data whenever required is possible in theory since it only depends on heightfield data, the logic that generates normal data from heightfield data is complex, as shown in A.2, and caching the normal data allows re-using existing normals in multiple frames. The texture type used to store normal values is selected as 2-component 8bit normalized floating point textures to have memory efficiency and the data range conversions to normal components as described above are applied on normal store and retrieve operations.

The advantages of the proposed normal generation pipeline are as the following:

1. Normal data can be sampled from vertex and fragment shaders using texture samplers that can perform fast linear interpolation and direct texel fetching.

2. Normal data generation does not require CPU intervention, all the operations are applied by the GPU, which also stores the heightfield normals.

3. Normal data size is reduced by using 2-component 8-bit-per-channel textures.

3.3.3 Generation of Heightfield Normals

Normal of a vertex or a point on a surface is a local geometric property. The regularly shaped heightfield structure can be directly used to calculate per-cell heightfield normal values since neighboring local mesh topology is available in heightfield data. Since the surface is only sampled at discrete locations and a parametric geometric model is not available for a complex heightfield, the normal vector of sampled points over heightfield can only be approximated using available discrete sample data. Some of the methods that can be used to approximate surface normals given a height cell sample point are:

- Central differences approximation [46]: The normals with the adjacent vertices (sample points) are computed (4 in total), the resulting normal vectors are summed and then normalized to unit length in order to yield a final normal vector for a cell point.
- Discrete differentiation [1]: The discrete differential is found on x and z axes by sampling 4 neighbor points and calculating the height differences in each axis. Y axis component is set to a pre-defined value. The resulting normal vector is then normalized. This method is computationally less expensive, but it does not use the center height sample and produce smoother normals.

The normal values are generated in a single shader logic which follows a GPU kernel approach, operating on all texels of an image. A single shared shader logic is implemented so that it can operate on both low and high resolution data with little computational overhead to scale to input size. The normal approximation method used is central differences approximation. Computations of normal vectors using shaders requires sampling 4 height values. In the case of computing edge cell normals, the values are sampled from a neighboring patch texture, since the required height sample maps to a region that is not stored in the current heightfield texture. The data flow diagram for normal generation is shown in Figure 3.4. The complete GLSL shader code that generates normal map for a given heightfield and its neighbors using the methods presented in this section is given in A.2.

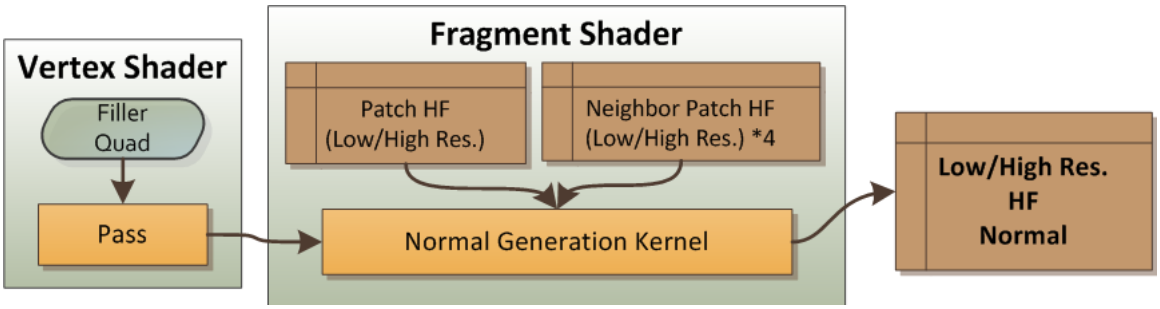


Figure 3.4: Data flow diagram for generating high/low resolution normals

Two neighboring patches must share the same normal data along their neighboring edge, as for the height values. A terrain patch that only uses low-resolution data may be a neighbour to a terrain patch that uses high-resolution data, or vice versa. Thus, the samples along the edges of a patch heightfield must be able to scale to high and low resolution textures. So, the sampling coordinates of neighboring height data depends on texture sizes of the neighbor heightfield. The sample values are rescaled to the current resolution cell size, when opposing neighbor data is of a resolution, to generate correctly approximated surface normals.

Another implementation detail is that when a texture is sampled using normalized floating point coordinates, OpenGL applies texture filtering to generate final data. However, sampling an exact texel is the underlying requirement to generate texture normals. Thus, the interpolated floating point sampling coordinates are converted into integer texture coordinates and the texel fetch shader operation is used to sample height values.

3.3.4 Collision Buffers

Collision buffers are data structures used to help the collision detection and compression methods as described in Section 4.1. It is basically a GPU frame buffer that maintains attachment of target and source heightfield textures and additional object collision data and collision detection results. All the textures associated with a collision buffer are high resolution textures, since it is targeted towards deformation simulation pipeline presented in this thesis.

Collision buffer is composed of

- a GPU frame buffer,
- an assigned terrain patch, from which high resolution compressed and high resolution uncompressed vertex displacement maps can be accessed.
- candidate collidable scene objects,
- object collision data textures,
- contact/penetration depth texture and other textures that stores the results of heightfield narrow-phase collision kernel, and
- GPU Timers that can be used for narrow-phase culling rigid body contact generation phase.

A collision buffer pool is created in application start-up, which stores predefined number of buffers. The buffers are distributed to terrain patches on collision detection time using this pool, as described in Section 4.1.1. The number of collision buffers limit the number of candidate colliding track patches and it should be set to support the number of colliding objects in the scene. The pool size setting may depend on the machine configuration and be used to limit the number of patches that can be simultaneously deformed. CPU-based simpler non-deforming heightfield intersection techniques can be applied on remaining patches that objects may collide with. The data stored in collision buffers are not permanent and may be overwritten by different patches in subsequent frames, since the buffers are re-distributed to candidate colliding terrain patches in each simulation step.

Chapter 4

Deformation Algorithms for Heightfields

This section presents collision detection, compression, decompression and erosion algorithms that are applied to heightfield structures in order to simulate physical interactions between heightfields and triangular objects. The aim of the methods in this section are:

- to update the vertex texture height (displacement maps) when an object collides with a heightfield cell,
- to animate compressed terrain cells after terrain is compressed,
- to define terrain and object material-based rules which affect collision detection, compression and decompression pipeline, and
- to generate data required for contact/collision data as used in rigid body physical simulation constraints.

4.1 Collision Detection and Heightfield Compression

In the proposed approach, collision detection is performed in two basic steps, which are broad phase collision detection and narrow phase collision detection. The narrow phase collision detection consists of two sub-steps, which are object collision data generation and per-cell heightfield collision detection. The final collision detection step applies compressibility logic and generates the compressed heightfield information. These steps are applied in sequence and each use the outputs of previous step. The final outputs of the collision detection phase are compressed high-resolution terrain height GPU textures and the rigid-body contact information for use by the rigid-body physical simulation step.

The broad phase collision detection operates on bounding primitives of objects and the bounding volume hierarchy of terrain and is managed by the CPU. The narrow phase collision detection and compression operates on GPU textures using GPU programs, thus can be highly data-parallelized on the GPU unit. The basic idea behind the per-cell collision detection is presented in Figure 4.1. Each cell is processed independently to find the collision and compression amount and the regions shaded in gray marks the cells in which object intersects the heightfield. Separating the narrow phase collision detection into two phases provides flexibility and increased efficiency. The per-cell collision detection kernel does not use z-buffer depth tests, so it can transfer results faster to the target textures. The z-buffer depth tests are applied only when needed, that is when creating object collision topologies. Object collision data is passed to exact per-cell collision detection step using 2D textures as intermediate data structures.

The two stages of narrow-phase collision detection cannot be merged together into a single GPU step(program). This is because of the fact that to be able customize terrain height update logics, such as introducing compression limiting, simple depth tests do not offer required programmability after output values in fragments are generated. This separation is one of the contributions of this thesis, which in turn allows complex programming of the terrain collision pipeline in the second collision pass.

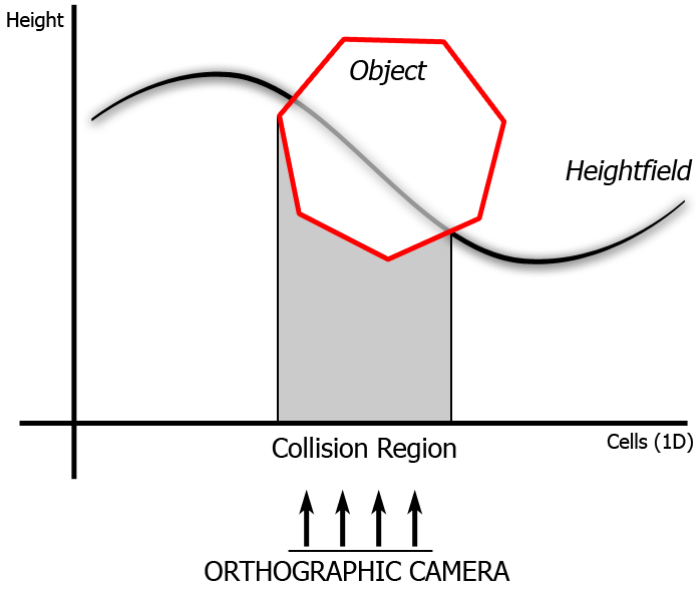


Figure 4.1: The identification of colliding cells on a 1D heightfield

The GPU methods described below take advantage of the representation of terrain height values as GPU textures. GPU textures are sampled directly to read height values at a given coordinate, instead of rendering a heightfield surface to generate per-cell topologies, as is the case with scene objects. Also, all the GPU textures used in the narrow phase collision detection phase are high resolution textures, which allow more precise physical simulation of terrain-object interactions, than the low-resolution textures which are targeted towards faster rendering.

4.1.1 Broad Phase Collision Detection

The aim of this phase, performed completely on the CPU, is to detect and cull candidate colliding terrain patch - object pairs. This step prevents running the exact narrow-phase collision detection for each object-terrain patch pair, thus increases the speed of the proposed system significantly. This step may produce candidate pairs even if they are not colliding, but it does not miss any collisions as long as object bounding volumes encapsulate the object meshes completely.

In order to perform broad phase collision detection, the axis-aligned bounding volume hierarchy of quadtree (see Section 3.2.1) and the simple bounding volumes

of each object are used. The object bounding volumes are not restricted to specific geometry types. As long as CPU-based boolean intersection tests can be performed between a geometric primitive and an axis-aligned box primitive as used by the quad-tree, this step can cull objects correctly.

The candidate collision pairs are placed into the collision buffers (see Section 3.3.4) which index the candidate terrain patches and store the list of candidate colliding scene objects for a specific terrain patch index. As candidate collision pairs are detected, the candidate scene object is registered into the scene object list of the collision buffer of the candidate terrain patch. In this step, if the patch does not have a reserved collision buffer, a new collision buffer is assigned for that terrain patch. This data management follows the fact that a patch may have many colliding object candidates at a given time. The rest of the algorithms are run on the active collision buffers with an assigned terrain patch.

The active collision buffer count cannot exceed the number of terrain patches in the system. Reducing the number of terrain patch indices used decreases the number of narrow-phase collision detection runs. Even though all the patches may be indexed, the objects will be most likely placed in one or very few (2-3) terrain patches in total since an object size is likely to be smaller than the size of terrain patches. This fact also shows a significantly reduction in the candidate colliding pair count.

4.1.2 Narrow Phase Object Collision Data Generation

Object collision data generation is the first step of narrow phase collision detection. The aim of this phase is to generate the object collision height, along with object collision material properties, that is later fed into the next phase. The output data is re-generated in each collision step, because the objects are assumed to be dynamic and the collision data is assumed to be invalidated after each step.

For each active collision buffer, the object collision data is generated from object triangular mesh, or its collision triangular mesh proxy, and stored within the collision buffer object as GPU textures. Other object properties can be

stored along collision data for processing in terrain compression phase as well. The object properties can be per-object or per-object-vertex. Object ID is such a per-object property, allowing detection of the exact colliding object (in case of a collision) when there are more than one registered candidate colliding objects to the same collision buffer. Another per-object property is the object mass, which can affect the compression amount of heightfield. The properties can be extended to other object features, such as object velocity and even object heat, as required by heightfield compression modeling.

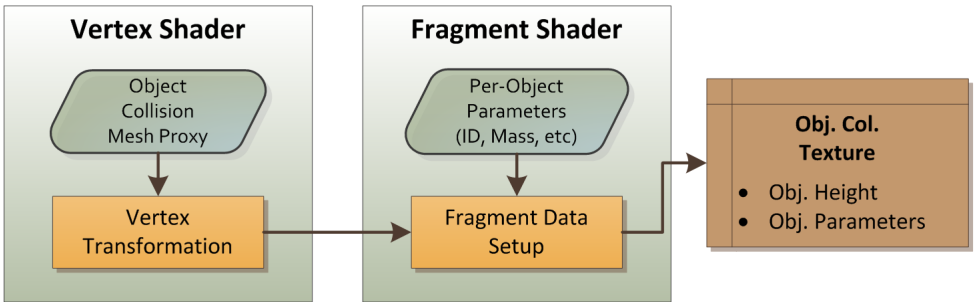


Figure 4.2: Object Collision Data Generator Program Data Flow Diagram

Every registered object is rendered sequentially to the object collision data target buffers of the collision buffer following the GPU program pipeline as shown in Figure 4.2. Since all the objects that can collide with a terrain patch are processed in a single batch with a single GPU program, the total process time is thus reduced. Only the vertex positions of the objects are processed (transformed) in this rendering step, which in turn creates simple and fast GPU vertex programs. It is possible to use lower or higher resolution object meshes as collision proxies in this phase and a proxy can either reduce number of vertices processed in a draw call or increase the collision detection precision. For example, this proxy can be used to set the heightfield-collidable mesh of a human body to its two feet only. The details of the implementation of this narrow-phase step are presented next.

The projection matrix is set to an orthogonal matrix. The camera is positioned on ground-level and is set to look upwards. The vertex shader is composed of a single transformation (multiplication) operation. The fragment shader linearly scales depth component in range [0,1] to world-space object height and also copies the Object ID into the target texture. The depth test is set to pass fragments by using comparison function $\leqslant$, so that the output produces the object height topology seen by looking up the vertical coordinate from the ground level and

other information related to the visible mesh (such as ID) is stored along with height information. This sub-step is automatically handled by the depth-testing functionality which occurs after fragment processing. The complete GLSL shader code of this phase is given in A.1.

4.1.3 Narrow Phase Exact Collision Detection and Compression

The aim of this second step of narrow-phase collision detection is to generate compressed terrain topology and collision contact data, given terrain and object collision data, including height topologies and collision material properties. This step can be implemented using a GPU kernel approach, allowing parallelism in per-cell calculations.

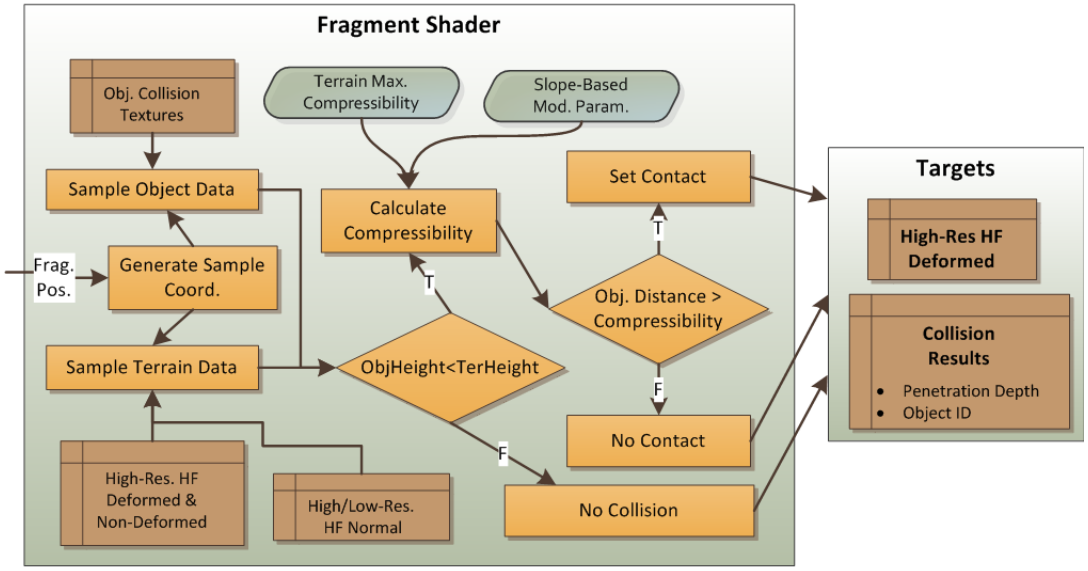


Figure 4.3: Collision and Compression Data Flow Diagram

For each heightfield cell, the GPU fragment-shader kernel of this step samples terrain and object collision data textures at the given grid position, and calculates the collision results and updated heightfield height values, as shown in Figure 4.3. A full-screen quad is rendered to generate fragments to sample each texel of both sources inside the fragment shader. Each terrain cell is processed only once in the GPU program for this phase. So, there is no associated over-draw penalty.

Terrain and object collision source textures storing height values have the same resolution and other heightfield collision material properties can be stored in lower resolution textures and per-sample vales can then be interpolated by the GPU. The values sampled from collision data textures correspond to the same sample point of the heightfield in world coordinates, so one-to-one correspondences in the processed texels can be maintained correctly.

To be able to model limited compressibility, the initial non-deformed terrain configuration needs to be maintained. Also, notice that high-resolution compressed (deformed) texture is both used as a texture source and texture target. Since a texture cannot be both a source target and a target in core OpenGL pipeline, two separate textures with identical properties are used to store the deformed height, as noted in Section 3.3.1.

The GPU program for this step has two basic targets for two purposes, as shown in Figure 4.3. The first target type is high resolution deformed heightfield texture, which is a member of terrain patch data structure. The second target type is collision result textures, which are the texture resources of the collision buffer that is being processed. Its content is the additional information required for physical simulation of scene objects. Deformed heightfield texture is persistent between simulation steps, yet the collision results are not persistent and can be updated by different patches in each phase. The internal processes shown in Figure 4.3 are described next.

Firstly, the sample coordinates for sampling heightfield and object collision data is generated from input fragment coordinates of the fragment shader. The sampled collision data includes undeformed-deformed terrain and object height values, which are used to detect whether a collision has occurred or not. No-collision case writes the input heightfield deformation factor without any modification and generates null collision result data.

The collision evaluation logic is based on calculating the amount of maximum terrain compression given the object and heightfield properties and checking whether current collision distance exceeds the maximum compressibility. If it does, data such as penetration depth and colliding object ID is written into collision result targets. Otherwise, the collision result targets can store that the

collision has occurred.

The current compression model is based on object mass, maximum terrain compressibility under unit weight and additional parameters that define the effect of terrain slope to compressibility. The maximum terrain compressibility is found using

$$maxCompression = comprUnitMass \times objMass_{Scaled} \times steepnessFactor$$

and

$$steepnessFactor = smoothstep(startSlope, saturateSlope, steepness)$$

, where $comprUnitMass$ is the maximum compression that a unit mass can cause, $objMass_{Scaled}$ is the object mass scaled between 0 and 1, sampled from object collision data textures, $startSlope$ is the minimum slope that the terrain becomes compressible and $saturateSlope$ is the minimum slope that the terrain is compressed to maximum under the same other terrain material properties. $smootstep$ is the function generating a smooth transition from 0 to 1 on the given limit values, scaling the previous term. Since mass is linearly proportional to gravity and the per unit-mass compression can be scaled with respect to gravity, the mass of object is substituted for the weight of object and compressibility per unit-mass is specified.

4.1.4 Narrow-Phase Culling for Collision Processing

Using the approach described above, once a terrain patch is selected as a collision candidate, the narrow-phase collision detection is run for that patch and the results are always read back to CPU for further processing. But, it is possible that, although the patch is a candidate for collision, the narrow phase collision detection phase does not detect any per-cell intersection. In other words, this case occurs when there are false positive candidate pairs after broad phase collision detection, that is, when the objects registered are not yet touching the ground.

The aim of the methods described in this section is to describe modifications to the algorithm above so that the application can detect whether there were any

intersections on the GPU kernel calculations and thus, to avoid unnecessary read-back of data to CPU and further sequential processing. The solution below does not require other data structures/GPU textures to be set up, the only overhead is the 1-channel height content duplication of two high resolution terrain textures, preferably using a GPU program.

This optimization uses the GPU occlusion query feature. This query allows retrieving the number of fragments that are written to frame buffer targets given a list of GPU commands. The fragment shader above is modified to discard fragments that do not generate an intersection between objects and the terrain. If the GPU occlusion query reports that no fragment has been generated by the collision and compression kernel, it implies that no terrain penetration has occurred and no height value was updated, thus preventing further processing of contact data.

However, using the methods described above without further modifications can result in de-synchronization between the two high resolution ping-pong textures of a terrain patch. The unmodified approach does not discard fragments when no collision occurs, so can keep the ping-pong textures synchronized after collisions occur. Yet, for the narrow phase culling to work correctly (to report 0 fragments generated on non-intersecting cases), we need to discard such fragments. This later prevents the propagation of the collided regions to the other texture object. It should be noted that this problem is a generic problem which arises in cases where two ping-pong textures are used and the transfer kernel from the source to destination texture discards fragments after a compression operation that uses the value in the source texture.

The situation that causes this side affect is presented in Figure 4.4. This figure shows two steps applied to an initially synchronized height texture pair T1 and T2. After the first step, intersecting regions, highlighted by gray regions, are modified on the destination texture T2. However, if the object intersects T2, which has now become the target source for the second step, only the object intersection regions are allowed to generate new fragments (compressed height data). The previously compressed regions of T2 cannot be transfered back to T1. The last row of the figure illustrates the de-synchronized regions of the two ping-pong compressed height textures.

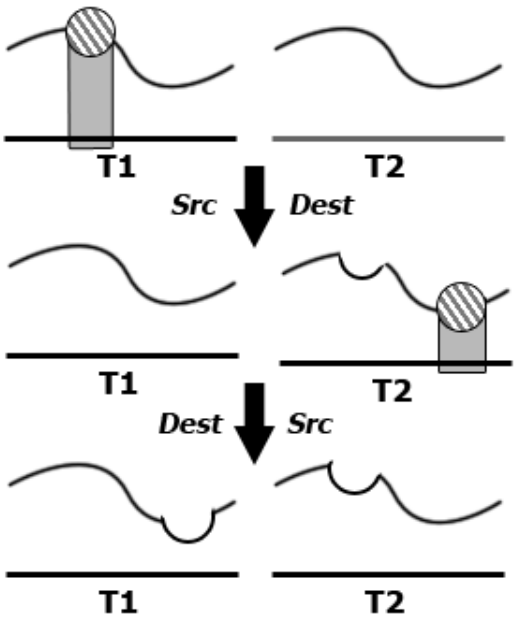


Figure 4.4: The Narrow-Phase Culling De-Synchronization Problem

The solution to this problem is to synchronize the two textures before running the collision program that can cull non-intersecting cases. The synchronization step replaces the output generation of non-culling collision program when there is no intersection and thus can correctly prevent the synchronization problem which, in turn, causes flickering on the rendered height data.

4.2 Decompression

In nature, some materials can deform back to their original state after deformations and most materials have internal stress and elasticity features that can help them resist to and recover from external forces. The aim of the algorithms presented in this section is to be able to animate deformed terrain regions to recover their height topology back to their exact or close initial undeformed state.

The methods described in this section are not based on transferring heightfield volumes. The methods can be considered as appearance-based models that aim to visualize viscosity-like properties of materials. If the ground material has high viscosity, the compressed regions are filled with new material from the

neighbourhood of the compressed region as time passes. However, the methods described here do not decrease the height, and so the volume of neighbouring regions. Another important assumption is that the initial uncompressed terrain region is assumed to be stable, so that convergence to the initial state is assumed to be convergence to a stable configuration. This problem is out of scope of the system described in this thesis, although it stands as a possible future extension.

A material de-compressibility parameter, which can correspond to *recRatio* in the following models, can be associated with each grid cell or set as a global terrain value. To implement per-cell approach, GPU textures and texture fetching in the shaders can be used. To implement per-terrain approach, GPU uniforms or, if the value is not expected to change over time, constant GPU variables can be used. Each decompression model below describes possible terrain material properties for that specific model.

This section will present three methods for decompression step. Each method presents different material and animation characteristics. The methods that are tagged as local operate on terrain cells without using any neighboring cell information. The presented erosion model uses the slope information per heightfield sample, thus it can capture the neighborhood topology of the sample point.

4.2.1 Local Linear-Speed Decompression Model

The local constant speed decompression model increases the height of compressed regions of a terrain patch to a target height using a linear animation speed. It uses two terrain material parameters, the recovery speed (*recSpeed*), and recovery ratio (*recRatio*). Recovery speed is expressed in terms of integer height value update per second and determines the speed at which the stable configuration is approached. Recovery ratio is used to derive the target height of a compressed region using the formula $h_{target} = h_{uncomp} - h_{maxComp} * recRatio$, where $0 \geqslant recRatio \geqslant 1$ and $h_{maxComp}$ is the height of maximum compression difference possible for a terrain cell.

In implementation phase, the animation needs to be correctly timed to variable frame rates of the application. All cells in a terrain patch are updated by a

constant amount in each iteration, $h_{stepSize}$. This variable is set by the application using the elapsed time since last update and the *recSpeed* variable.

The new height of a terrain height grid cell is then found using the following formula:

$$h_{new} = \begin{cases} h_{old} \ , & h \geqslant h_{target} \\ \min(h_{old} + h_{stepSize}, h_{target}) \ , & else \end{cases}$$

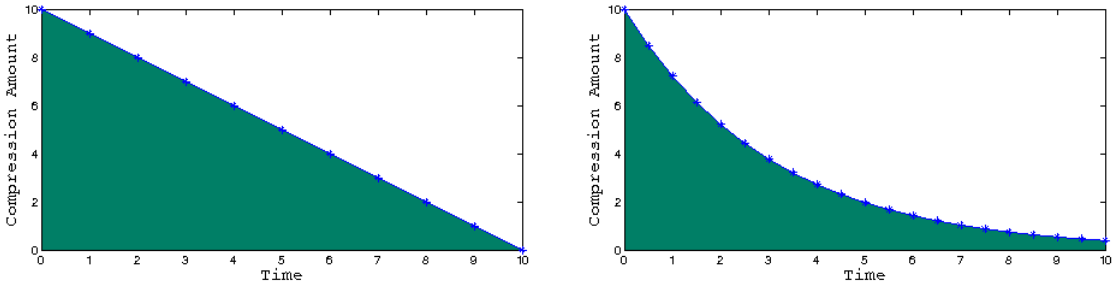
In this approach, small deformations are fully decompressed earlier than larger deformations. De-compression phase does not affect regions that are deformed less than set by the recovery ratio.

This model can be extended by another terrain material property which can modify the recovery speed as the following: $h_{stepSize+New} = h_{stepSize} * perSample$. This property can be stored on a small texture for the whole terrain, while per-sample attributes are interpolated by the GPU.

4.2.2 Local Exponential-Speed Decompression Model

The difference of this decomposition model with respect to the previous model is that the deformation recovery speed is dependent on dynamic per-sample(cell) compression amount. The shape recovery curves of these models are shown in-Figure 4.5. This method also uses two terrain material parameters, the recovery speed (*recSpeed*), and recovery ratio (*recRatio*). The meaning and usage of recovery ratio variable is the same as the previous model. Recovery speed is used to derive the ratio of terrain material to be decompressed (*stepRatio*) at an iteration of decompression program. In the implementation, *stepRatio* is found using explicit first-order Euler integration of the height recovery curve shown in Figure 4.5b with time steps of size 0.01.

The new height of a terrain height grid cell is found using the following formula:



(a) Linear Speed Decompression Model (b) Exponential Speed Decompression Model

Figure 4.5: Local Decompression Models

$$h_{new} = \begin{cases} h_{old} \ , & h \geqslant h_{target} \\ h_{old} + stepRatio * (h_{target} - h_{old}) \ , & else \end{cases}$$

This model allows faster recovery of height values in regions that have been compressed more. Although the lesser compressed regions approach to target values faster, the convergence time difference between varying deformations on heightfield is smaller.

4.2.3 Erosion Decompression Model

This model uses the normal at the deformed height sample along with the height and compression information. The aim of this model is to limit the slope of compressed regions, while the above models specify recovery limits in terms of absolute height. The recovery ratio (*recRatio*) parameter associated with this model specifies the limiting angle which separates heightfield cells that will be decompressed or not. The recovery speed (*recSpeed*) can either follow linear-speed or exponential-speed model.

The new height of a terrain height grid cell is found using the following formula:

$$h_{new} = \begin{cases} h_{old} \ , & slope \leqslant recRatio \\ h_{old} + h_{stepSize} \ , & else \ \ (using \ linear \ speed) \end{cases}$$

Since this model first fills in the compressed regions that have higher local slope, it is able to create the perception of an erosion, although no cell is lowered because of a volume transfer.

As a summary, the models above are computationally cheap and follow different approaches for animating decompressions. The methods are likely to perform at the same speed, since the costs per sample value are similar between different models. The models can be further mixed together, such as implementing separate decompression recovery ratios for absolute compression amount and maximum decompression slope and separate decompression recovery speeds for both linear and exponential speed models. This will further enhance the material-specific behaviour of compressed regions. As noted previously, the parameters can be constant for whole terrain, or can be updated in a per-cell approach, which allows mixing render properties and decompression material properties of terrain regions.

Chapter 5

Physical Simulation of Rigid Bodies

The interaction between a heightfield and scene objects is a two-fold process; the objects deform the heightfield and the heightfield applies resisting forces/ constraints that prevent objects from passing through the heightfield, falling down. Chapter 4 described the simulation of heightfield deformations, that is, object to heightfield interactions and additional methods for animating deformations on heightfields. This section describes and discusses methods that can model heightfield to object interactions, using the data generated in the previous phase. The objects are assumed to be rigid and non-deforming.

In the proposed system, the physical impact of heightfield to scene objects is solved through an integrated rigid body physics simulation which applies basic Newton dynamics. The only requirement to add terrain to object interactions on top of correctly generated contact data is a contact constraint solver for rigid bodies. This chapter presents the basic physical simulation engine wrapper layer and the conversion of terrain-object collision data to rigid-body simulation contact data, including surface parameters.

5.1 Physical Simulation Engine Wrapper Layer

Since building a complete and working 3D rigid body physics engine is a task with many practical and implementation details and extending and implementing a dynamic physics solver is not a target in this thesis, an existing rigid body physics engine, Open Dynamics Engine (ODE) [8], is integrated to enable physical simulation of rigid bodies.

To support the complete integration of the graphics, collision and physical data, a C++ wrapper over ODE is implemented. The Open Dynamic Engine wrapper and graphics integration implementation is distributed with a free software license [1] as an extension over Open Rendering Engine (OpenREng) developed as a part of this thesis (see Section 7.1.2). The source code can be used by other projects that wish to use an integrated physical simulation framework with OpenREng.

The available wrapper further allows a future replacement of the underlying physics engine with other available physics engines, so that the implementation is not targeted towards a single physics engine. The reasons that ODE has been chosen as the primary underlying engine are listed as the following:

- ODE supports separating collision detection and simulation of constrainted environments, as required by the proposed system.
- ODE has been used previously in state-of-art academic works [49, 42].
- The project is active, mostly stable and supported through its community.

The features of the physical simulation layer implementation for this thesis include the following:

- A fully object-oriented and documented C++ Physics API (Existing ODE interface is targeted towards C, although ODE is implemented internally in C++),

[1] `http://openreng.svn.sourceforge.net/viewvc/openreng/trunk/tools/Phy/`

- Wrappers over basic physics concepts in ODE (will be described later),
- Automated scaling between application coordinate and physics coordinate system, since the physical calculations may require the world fit into more appropriate scales to be able to prevent floating point calculation errors, and
- Automated synchronization between rigid bodies and renderable meshes during physical simulation.

The object-oriented wrappers included in the implementation encapsulate internal ODE object pointers and operate over their data through class methods, while also extending their functionality as seen required. The wrappers cover most of ODE structures and currently include the following types:

- rigid body auto-sleeping configuration interface, both per-body and per-world-defaults,
- rigid body damping configuration interface, both per-body and per-world-defaults,
- rigid body substance interface (storing mass, center of gravity and moment of inertia of a rigid body),
- rigid body class (storing information such as body position-orientation, body substance, attached joints, attached physics geometries and including interfaces related to force and torque insertion),
- physics body and render nodes spatial linker (allowing automatic synchronisation of position and orientation from physics-world bodies to the scene nodes used for rendering),
- primitive geometry classes (allowing attachment of geometries to rigid bodies or to the static physics world, for use in collision detection within the physics engine),
- geometry space classes (storing a list of geometric primitives or other geometry spaces, thus allowing a hierarchy of spaces for internal rigid body collision detection),

- world class (managing world-specific stepping/integration settings and geometry spaces within a world),
- simulator class (managing internal stable stepping/integration of the worlds by a shared continuous stepping interface),
- contact surface parameters (storing information such as friction constants, bounciness, error reduction parameters, friction approximations, friction directions, etc),
- contact constraints between rigid bodies (storing contact surface parameters and contact spatial information),
- customizable collision callback interface, and
- a debug renderer that can render the requested geometric entities appropriately, and contact joints as a triad.

The geometric primitive entities that ODE wrappers include are spheres, boxes, cylinders, capsules (swept spheres), half-spaces and rays. ODE can internally detect collision between these primitive geometry types and call the assigned callbacks for generation of specific contact data by the application. These physics geometries are not used by the object-heightfield collision detection and deformation pipeline, as described in Chapter 4. The effect of the simulation of these physics geometric primitives is visible on the implementation of this thesis when two objects in the simulation interact with each other.

5.2 Generating Contacts from Collision and Heightfield Data

To generate contact information from colliding points, per-cell collision detection results generated by the GPU are read back to the application for further processing and passing into the physics simulation engine.

The data passed to CPU are read from high-resolution GPU textures and the content can be listed as:

- the final (compressed) height,
- the ID of the object that has compressed the terrain cell (if any),
- the object penetration height into a fully compressed height, and
- additional terrain material properties, if sampled/processed by the GPU.

The rigid body contact joint data for a grid needs to be generated when the object penetration height is set to larger than 0. The related contact data components, their brief descriptions and generation details are as follows:

- **Contact position:** *3D vector representing the contact position between objects.*

  X and Z components are generated from patch position and active cell's grid coordinate. Y component (height) is generated from active cell's terrain height data, as read from GPU.

- **Contact normal:** *3D vector representing the direction in which the height-field contact response needs to be generated.*

  Contact normal can be generated from high-resolution heightfield texture using neighboring heights, or directly be read from GPU high-resolution normal texture, which can affect the run-time efficiency. The best option is likely to depend on machine configuration, basically GPU to CPU read-back performance and CPU clock speed for the normal calculation maths.

- **Object-terrain contact depth:** *A single scalar value holding the penetration amount, so that if the colliding rigid body is moved along contact normal direction at depth distance, the object will no longer collide with the terrain.*

  Depth is generated from active cell's collision depth data, as read from GPU. Note that this data is directed towards the up-axis of the heightfield, while the data description requires the depth to be along the normal vector on a height sample. The data along the normal axis can be generated with an additional full-sizde pass on the patch, marching along the normal vector of a sample and finding the intersection point and the intersection distance.

This approach has not been implemented and tested, since we have noticed that the simple approach can be used for a stable simulation.

- **Colliding rigid bodies:** *The two rigid body ID's that denote the bodies colliding with each other.*

  The first collision object is set to the object that collides the region and it is read from active cell's colliding object ID, which points to a specific object in the scene. The second object is the terrain, which is implemented as a static world component, which cannot be physically affected from collisions.

- **Contact surface parameters:** *The parameters include friction settings, bounciness, error reduction and softness parameters, etc.*

  Per-cell contact surface parameters can be read back from GPU textures or can be sampled by the CPU from terrain attribute images on the CPU. The effects that can be achieved using variations over surface parameters include friction amount and bounciness.

- **Contact first friction direction:** (Optional) *3D vector perpendicular to contact normal. Second friction direction is the unique vector perpendicular to both this vector and the normal vector. If first friction direction is not set, its value in physics solver is not predictable.*

  Currently, the friction direction is not set.

Figure A.8 shows the objects resting on terrain on the upper row and the contact points between the objects and the heightfield, along with object bounding boxes, on the lower row. Notice that the objects are resting on heightfield are in contact with the heightfield in multiple cells. Since the intersections are allowed first and fixed later, stable resting is only possible when there exists shallow contacts that can balance the object on the heightfield. The exact contact points and contact normals are magnified in Figure A.9. This figure shows the wire-frame overlay rendering of heightfield in high-resolution (physical simulation) precision, so the matching between grids and contact positions can be better observed.

Chapter 6

Heightfield Visualization

This section describes methods for visualizing the heightfield data efficiently using GPU hardware. The presented methods include a multi-resolution level-of-detail and culling system on CPU for low-resolution heightfield data and a unified GPU shading approach that can render both non-deformed and deformed regions, blending high and low resolution heightfield data as necessary. The unified shading system is built to support different deformation blending approaches with varying complexity and efficiency, thus the system is extended to support rendering level-of-detail for deformed regions as well.

6.1 Low-Resolution Level-Of-Detail and Culling

Low-resolution heightfield data is sufficient alone to generate renderings of basic heightfield topology. Since targeted low-resolution heightfields are not small scale (ex: 2048x2048), the vertex count of a complete heightfield can exceed orders of magnitudes of 2^{10}, which creates a challenging rendering task that has been studied for decades (Section 2.1.3). The aim of the methods presented in this section is to prepare pre-frame render data on CPU as efficiently as possible in the proposed system. Since details of rendered parts close to viewer are more visible and distant regions occupy smaller screen space, level-of-detail systems

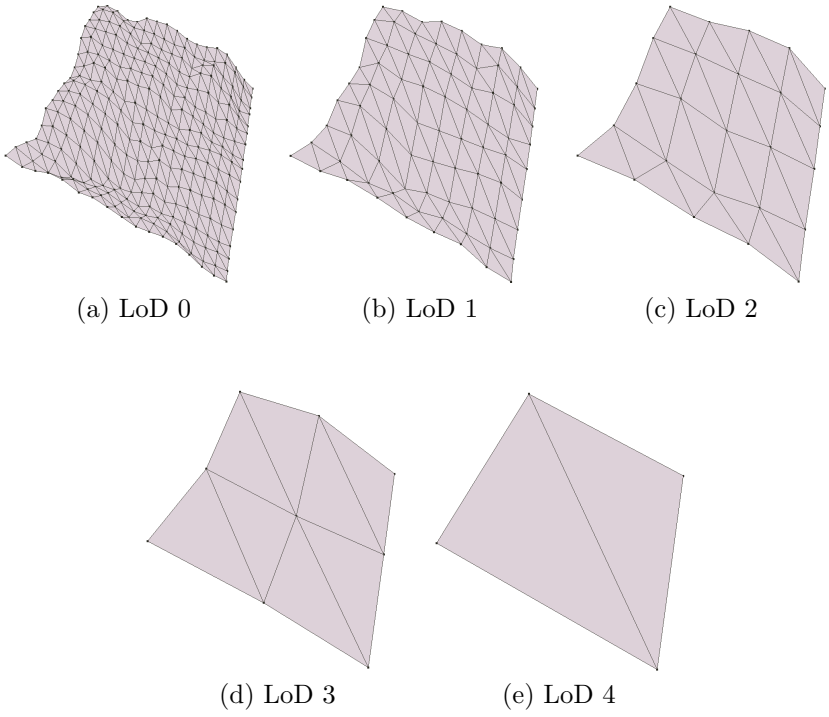
for heightfields can be employed to speed-up rendering significantly. Other optimizations described in this section target avoiding processing non-visible parts of heightfield during rendering and fall under culling schemes for heightfields.

6.1.1 Geo-Mipmapping Level-Of-Detail

The basic algorithm that is chosen for low-resolution level-of-detail implementation is geometric mipmapping (geo-mipmapping) ([10]). Geo-mipmapping is a simple multi-resolution level-of-detail approach and works similar to texture mipmapping. With this approach, given a heightfield texture, each refinement of a heightfield block reduces vertex count to one-forth, applying a coarser sampling of height values over the heightfield, as shown in Figure 6.1. Notice that this approach does not offer an optimized triangulation over a heightfield surface and the number of layers for a block are limited by the patch size, while there exist other methods or models, such as those based on triangulated irregular networks (TIN's) and continuous level-of-detail schemes, that aim to provide more optimal approximations of heightfield surface. This characteristic of the selected base method is compensated by the fact that GPUs can handle large number of vertices faster than a possible management overhead of a complex level-of-detail systems on the CPU.

To generate a mesh from a heightfield data using geo-mipmapping technique, dynamic indexing over the heightfield vertex data is used in the proposed system. The index data is dependent on level-of-detail configuration of a patch and the generation logic is presented in Section 6.1.2. Generating a specific detail layer is thus mapped to constructing correct indexing over the list of heightfield samples, reducing the number of samples/vertices used. Note that the sampled heights at the indexed vertices remain constant since the underlying data is not modified to generate new layers.

Since mipmapping approach itself does not offer a tessellation of a heightfield with respect to the viewing distance, the decomposition of large-scale heightfield into smaller blocks is necessary. Since the heightfield management proposed in this thesis is already decomposed into regular sized terrain patch structures,



(a) LoD 0 (b) LoD 1 (c) LoD 2

(d) LoD 3 (e) LoD 4

Figure 6.1: Geo-mipmapped layers of a sample heightfield block

geo-mipmapping is directly applied to low resolution heightfield blocks stored in terrain patches. The vertex data, set up as a dense, regular and ordered list of 2D vertices on a grid structure, is shared between all the patch blocks on terrain. The third component of 3D position of a heightfield vertex (height) is set by a heightfield texture lookup at render time.

Given a geo-mipmapped block (terrain patch) of size $(2^{patchSize} + 1) \times (2^{patchSize} + 1)$, the number of discrete detail layers is $patchSize$. The active detail layer depends on the distance of the viewpoint to a terrain patch. Screen-space error that occurs when changing between sequent detail layers is another parameter that can be considered while selecting the suitable layer given a viewpoint.

The active layer selection metric that has been used in the implementation follows the method proposed in [10]. This method uses maximum screen space error on transition between layers as a parameter and it generates a pre-computed

viewpoint distance map for each block that is later used to find the active level-of-detail configuration in each rendered frame. The maximum screen-space error is computed from maximum world-space error, which is computed when the heightfield block data is set. In this phase, it is assumed that the projection is perspective, so that screen-space error is reduced as the heightfield block gets more distant. It is also assumed that viewpoint is at the same height level, looking horizontally to the heightfield range, which generates maximum screen-space errors given a distance from a heightfield. This approximate error metric is used to find the threshold view distance positions so that a change between two sequent layers of geo-mipmaps produces a screen-space error as close to, but not exceeding the given screen-space error. The calculation of threshold distances are done using the following formula, as presented in [10]:

$$D_n = |\delta| \cdot \frac{nc \cdot v_{res}}{2 \cdot |t| \cdot \tau}$$

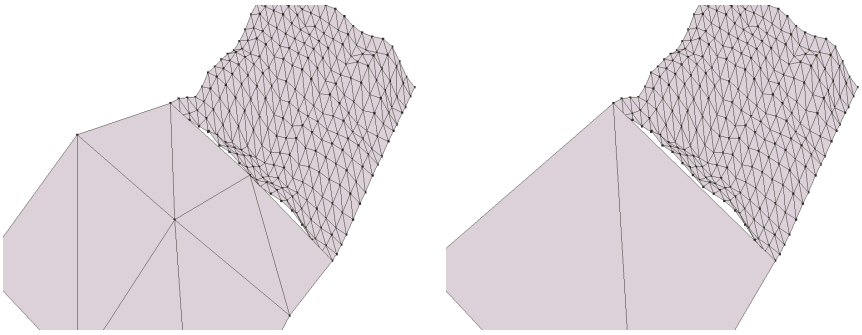
In this formula, $|\delta_n|$ is the maximum world-space error when transitioning from nth layer to $n+1$th layer, nc is the camera near clipping plane world-space distance, v_{res} is the vertical resolution of the screen, t is the top coordinate of the near clipping plane, and τ is the maximum allowed screen space error.

Advantages of the extended geo-mipmapping approach in the system proposed in this thesis are:

- All detail layers use the same underlying heightfield data, no new vertex data is generated for different layers.
- Using a complex index generation logic, there is no restriction on the active level of detail layer differences between neighboring patches.
- The underlying patch and quadtree structures can be directly mapped to geo-mipmapping methods.
- The lightweight (2 coordinate) vertex data that is used to render the patches is shared between patches. This allows rendering all patches without changing vertex buffer state of GPU hardware and reducing the amount of memory required to store the heightfield 3D vertex positions.

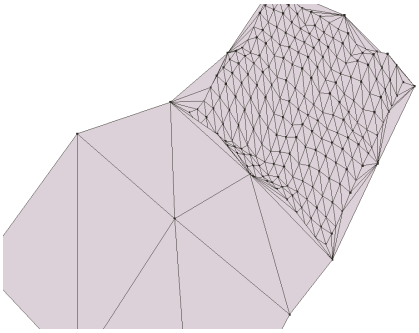
6.1.2 Generating Geo-Mipmapped Index Data for Heightfield Blocks

While generation of an index list for a regular heightfield block which does not need to connect to neighboring heightfield blocks is a straightforward task, the need to seamlessly merge neighboring heightfield block stands as a challenge in terrain level-of-detail systems. Without taking care of merging neighboring vertices, the cracks (T-Verties) between neighboring blocks will be visible, as shown in Figure 6.2a and Figure 6.2b.

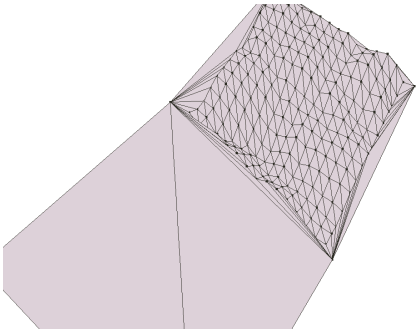


(a) T-Vertices (3 layer dif.)

(b) T-Vertices (4 layer dif.)



(c) Avoidance (3 layer dif.)



(d) Avoidance (4 layer dif.)

Figure 6.2: Visible and avoided T-Vertices between different detail layers

To efficiently avoid T-vertices between different heightfield blocks, one can either update the indexing scheme in lower-detailed layer or the higher-detailed layer. Refinement on higher-detailed layer is based on removing vertices along the edge of the heightfield block so that the remaining vertices match to neighboring block's edge. On the other hand, refinement on lower-detailed layer requires

inserting vertices along the edge of the block. Removing the vertices from high-detailed layer affects smaller regions on heightfield and this approach is chosen as the base approach to avoid the T-vertices between neighboring heightfield blocks.

The indexing method used to render a heightfield block is selected as triangle strips. It is known that using a single triangle strip index data and degenerate triangles within the indexing, any vertex topology can be created. Recent GPU hardware can also discard degenerate triangles, reducing the number of triangles processed internally. In the implementation, each block of heightfield is rendered as a single mesh using a dynamic index buffer that is composed of a single triangle strip index data. Mapping heightfield blocks to a single renderable mesh increases rendering efficiency of the heightfield.

The proposed/implemented index buffer generator is independent of other structures used in the system. The required data to generate an index buffer are the highest-detail layer heightfield size, the self active detail level and four neighboring block's active detail layer. The target indexing data is decomposed into five regions; four of them are the single-cell sized border buffer regions and one is the internal region, which is not affected from neighbor's active detail levels. Degenerate triangles are more commonly inserted in regions on the patch border. Figure 6.3 shows the indexing data generated for a heightfield block, in which active layer for self is a high-detail layer and each neighboring block has a different active layer. Namely, if level-of-detail layer of the patch shown in the figure is k, left neighbor's layer is $k-1$, right neighbor's layer is $k-2$, upper neighbor's layer is $k-3$ and lower neighbor's layer is $k-4$ Note that the inner region is shown in darker shading.

If terrain sub-patch structures are used for rendering, the index generator receives a list of sub-patches inside a terrain patch and correctly skips filling in the grids that correspond to a sub-patch structure within. Also, the sub-patch rendering pipeline uses shared index and vertex buffers that renders a high-resolution terrain sub-patch at full resolution. No cracks / t-vertices appear along the edge of sub-patches since

- sub-patches do not have internal level-of-detail configuration,

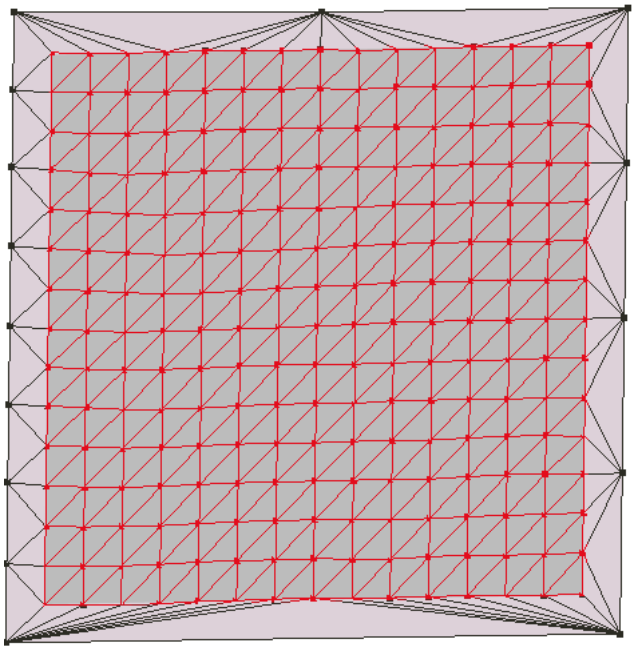


Figure 6.3: Complex triangle strips data of a heightfield block

- sub-patches are rendered only when their owner patch is rendered in the highest level-of-detail layer, and
- the vertices on the edge between a sub-patch and its owner patch share the same vertex position.

Using unsigned integer height values also help in this phase to avoid z-buffer fighting that may appear when using dynamically updated floating point data types.

The logic that generates the complete index list for a terrain patch that can scale itself to sub-patches and varying level of detail difference is implemented as a part of this thesis. It has been observed that index data generation time does not affect the final frame rate, since the camera is expected to move relatively slowly, which creates higher coherence between frames, and the LOD configuration of patches does not change frequently. Caching frequent variables inside the index data generator and using shared index buffers for multiple frames has helped to achieve the higher performance of the index generator. Figure 6.1 shows the triangulation performance of the index data generator implementation. The data has been captured on a real-time fly-through over a terrain of size 1025×1025,

with no sub-patch gaps. Since the camera was dynamic, many configurations of patch level-of-details had been activated through the experiment. The simple experiments shows that as the terrain patch size increase, the ratio of degenerate triangles decreases. This follows the fact that most of the triangles lie in the inner region of a terrain patch, as seen in Figure 6.3.

Table 6.1: Triangulation performance of the index data generator implementation

Patch Size	**Number of buffers generated**	**Total number of indices**	**Ratio of degenerate indices**
17×17	5595	784145	9.87%
33×33	5408	1665051	6.52%
65×65	3687	4346407	3.38%
129×129	1500	8235200	1.56%

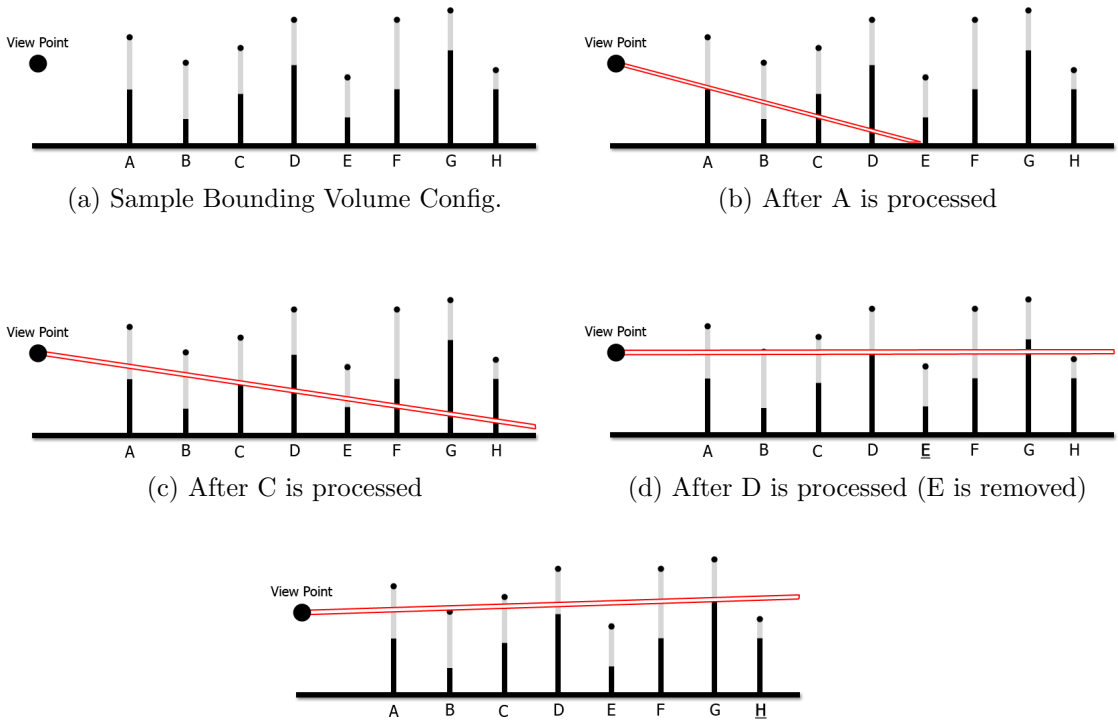
6.1.3 Terrain Patch and Primitive Culling Optimizations

The AABB volume hierarchy that is stored in terrain quadtree, as described in Section 3.2.1, allows culling invisible patches using camera frustum geometry and terrain patch AABB before sending the patch render data to GPU. Notice that this step cannot apply culling to individual triangles within a patch geometry. Figure A.6 shows the frustum culled regions on a terrain. In this image, the complete terrain is rendered in wire-frame mode and visible patches from a different view-point in the scene are also rendered as filled heightfield blocks.

Another optimization aims to exploit early z-buffer culling feature of recent GPU hardware. This feature allows GPU to discard all fragments of a triangle primitive before they are generated by comparing triangle vertex depth values to frame depth buffer values. To take advantage of this feature, GPU should render closer objects before farther objects. Therefore, in the current system, terrain patches are sorted in front-to-back order with respect to the squared distance to the viewpoint, before they are sent in-order to the GPU for drawing.

In addition to frustum culling, occlusion culling can be applied to prevent drawing of patches that are completely occluded by patches closer to viewpoint. The culling algorithm can be based on conditional rendering with occlusion queries on GPU, while the geometries that are rendered are simple AABB's,

or the algorithm can be run on the GPU by maintaining a visibility horizon structure, as discussed in [29]. The existing patch AABB's can be used as efficient occluder and occludee geometries. While an AABB sometimes provides a crude approximation of the underlying mesh geometry, fast rejection of some of the invisible patches should be the target of the culling operation. An important observation is that the regions below the AABB of a patch can be assumed to be fully complete, thus the occluders are likely to occupy a large screen-space, especially those closer to and above of the viewpoint. This approximated approach follows the robust and simplistic LOD and index data set-up on the CPU.



(a) Sample Bounding Volume Config.

(b) After A is processed

(c) After C is processed

(d) After D is processed (E is removed)

(e) After G is processed (H is removed)

Figure 6.4: Patch-Bound Occlusion Culling in 1D

To present the basic idea behind the occlusion culling method that has been developed, a 1D patch configuration, with 8 distance-sorted patches is shown in Figure 6.4. The light shaded upper regions of the patches denote the bounds of these patches in 1D. In this 1D case of the proposed algorithm, a single active ray from the viewpoint is maintained for occlusion culling purpose. The patches

are processed in front-to-back order with respect to the viewpoint. After the first patch A is processed, the active ray is set to pass from the lower bound of the bounding box of A. When patch B is processed, since the ray passes through the bounding box, B cannot be culled and the active occlusion ray cannot be updated. Intersection of patch C and the ray appears below the lower patch bound of C, so the active ray is updated at this step. The same argument applies also to patch D. Patch E upper bound is lower than the ray height at patch E, so E is culled by the previously processed patches. Later, patch G also updates the occlusion ray and patch H remains below the final occlusion ray, which marks it as the second occluded patch.

To extend the algorithm above to 3D case on CPU, the 6 vertices of the AABB of a patch need to be transformed to screen-space. The transformed vertices are then either processed using scan-conversion of transformed vertices to a 1D array of screen width, or inserted into a -preferably- sorted list of horizon vertices, updating the occlusion data as required in both cases. The bookkeeping operations should be implemented as time-optimized as possible. The GPU implementation requires using a separate frame buffer for storing occlusion results.

Based on the available patch structures, the method described above requires no additional preprocessing time to employ a fast and efficient visibility detection method. When the camera is positioned on the ground level and a large heightfield is used, it is highly likely that there will be hills on the heightfield which occludes many of the patches behind. The ratio of occluded patches to non-occluded patches can be high when regions with hills / mountains are rendered close to the viewpoint, and the additional book-keeping time is expected to be small, offering a way to optimize the render speed of the given heightfield management system. However, the presented approaches could not be implemented in the sample application because of time constraints and thus, results related to this approach cannot be provided. For the basic non-deformed patch rendering performances obtained without patch occlusion culling, along with the statistics of rendered patch counts, you can consult to Table 7.9.

6.2 GPU Shading for Visualization

The previous section described how the rendering data sent for rendering to GPU is processed and optimized for efficiency. This section describes the shader techniques that are proposed and implemented to complete the rendering pipeline of the proposed terrain system.

The GPU shading methods in this section are based on the conventional 3D rendering pipelines, which use depth buffers and process the vertex and triangle information through vertex and fragment shaders. Although new shader stages are available in the recently published specifications, such as geometry shaders and tessellation shaders, those stages are not activated and used by the shading methods proposed in this section.

On top of the data structures described in Chapter 3, a unified shading system is implemented for efficient visualization of heightfield deformations. The overview of this unified shader program is separated into per-vertex and per-fragment GPU pipelines and the data flow charts of these shading units are shown in Figures 6.5 and 6.6.

Branches on deformation rendering techniques are implemented as separate shading programs to speed-up GPU program execution, which basically runs in SIMD (single instruction multiple data) mode. Rendering of non-deformed regions follows simpler paths in shaders, without calculating deformation parameters and branch-specific variables.

The following section focuses on specific processes within the unified renderer, such as generation of vertex positions, texturing, lighting, enhancements on deformed regions, and specific paths for two-step and single-step high/low resolution heightfield blending techniques. The common processes shared between different blending techniques are described first and the deformation shading methods are described later. Finally, a deformation shading level of detail system that can scale rendering speed and quality with respect to viewing distance is presented. The complete unified vertex and fragment shading code written in OpenGL Shading Language, including all the routines for the methods described below, is given in Section A.3.

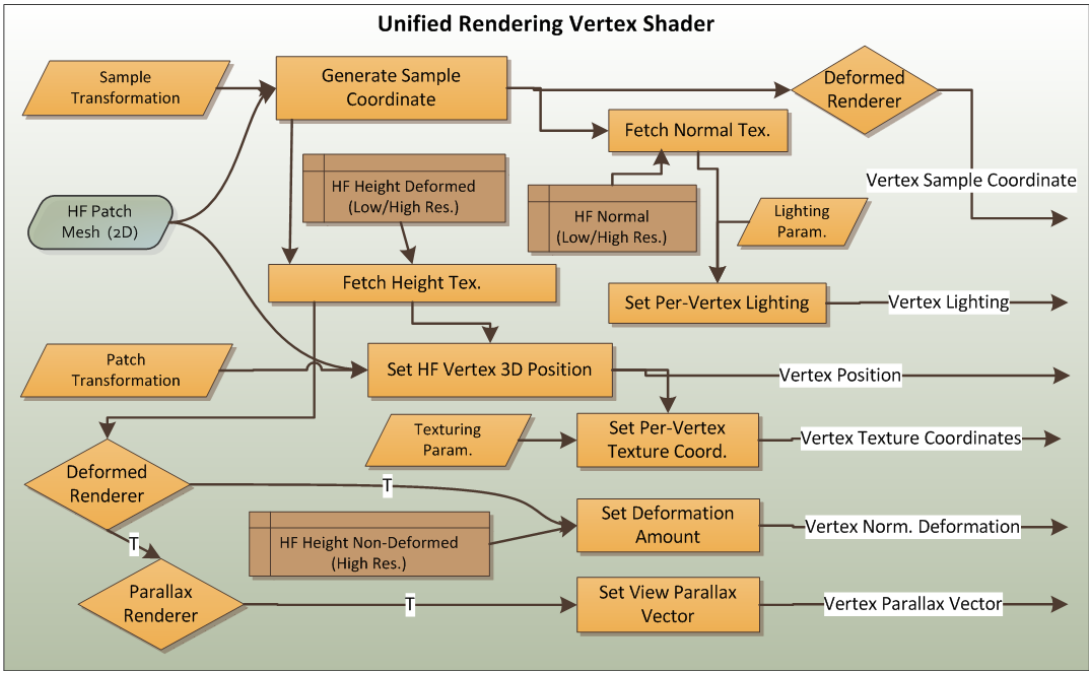


Figure 6.5: Unified Rendering Vertex Shader

6.2.1 Generation of Vertex Geometry

The vertex geometry described in this section includes the vertex position and vertex normal vector. The world-space position of a heightfield vertex as processed by the vertex shader is calculated using only the input heightfield mesh, per-patch vertex transformation vectors and the texture that holds height sample data. The mesh data is composed of a single static, constant vertex buffer indexed by a dynamic index buffer, as described in Section 6.1.2. The shared mesh vertices form a regular uniform square grid on the XZ plane, with the limit coordinates normalized to $[-1, +1]$. The Y coordinate of an input vertex is set by the vertex shader after sampling the height value using a normalized texture coordinate (in range [0,1]) generated from the XZ coordinates of the vertex (in range [-1,+1]) using the formula $heightTexCoord = vertexPosition.xy * 0.5 + 0.5$.The input 2D vertex is transformed and uniformly scaled to generate XZ components of world-space 3D vertex using the formula $vertexPosisition.xz = vertexPosition.xy * patchScale + patchTranslate$, so that the input unit square mesh maps the world-space region of the patch on XZ plane. The 3-component patch translation data, $patchScale$ and $patchTranslate$ are managed by the terrain patch that is being rendered. Notice that the approach described above keeps

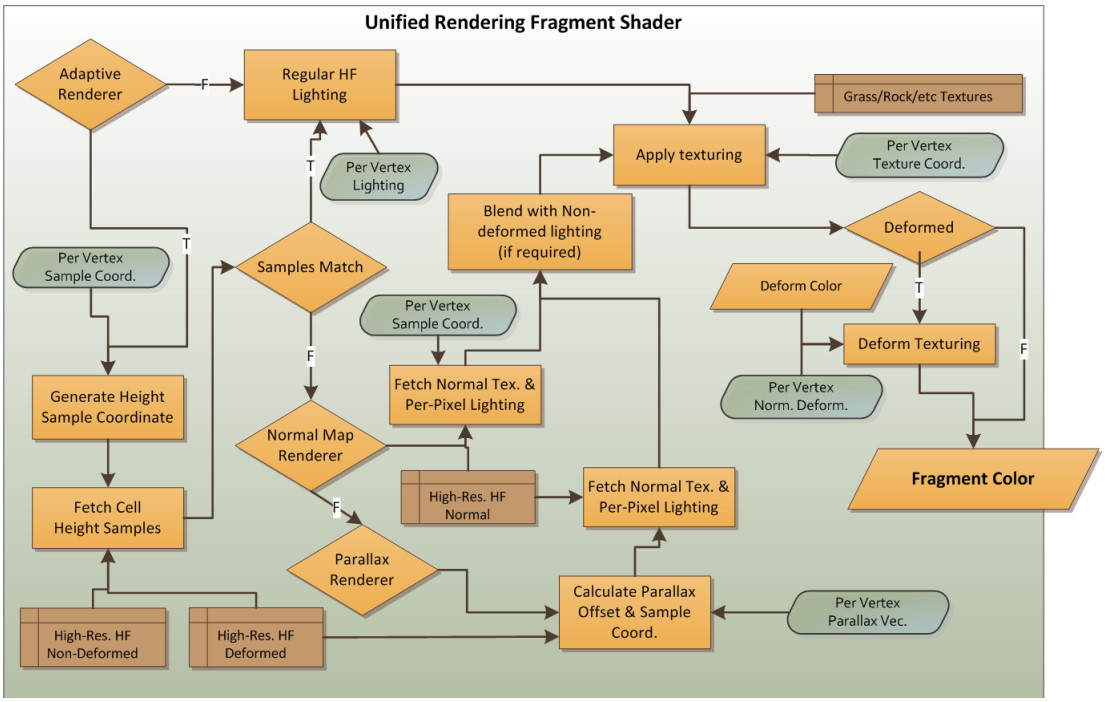


Figure 6.6: Unified Rendering Fragment Shader

the memory bandwidth requirement for processing a heightfield input mesh to minimum.

As described in Section 3.3.2, two-component normalized floating point GPU textures are used as the normal vector data source for the heightfield blocks. Normal texture maps, which can be low or high-resolution, are accessible by both vertex shader and by fragment shader when required by the active deformation specific shading. The high resolution normal maps are bound to shading programs only when the terrain patch is deformed, thus the patch has high resolution normal map data. In the vertex shader, the normal texture map is sampled with texel fetch operation, using integer texture coordinates, in order to retrieve per-cell normal vectors without filtering interpolations applied. The filtering operations can produce artifacts when the normal resolution changes for a patch render step. In the fragment shader, if a deformed patch is rendered, higher resolution normal map is used to shade the deformed cells. The sampling in fragment shader uses hardware accelerated linear magnification filtering to be able to produce smoothly varying normal values over the heightfield surface. As a result, two different normal texture sampling approaches are used in a single GPU program. While it is not possible to modify the filtering method of a bounded texture in different

shading units (vertex or fragment shaders), it is possible to mix direct texel fetch operations with filtered texture sampling. Using the approaches described in this paragraph and in Section 3.3.2, surface normal popping artifacts can be avoided to some extent when there is a switch between different resolution normal maps and the normals of sample points that are shared between neighboring patches are set to same values.

6.2.2 Heightfield Texturing and Lighting

To generate a realistic output of heightfield data, applying textures to heightfield triangles (painting) is necessary. Realistic texturing of terrain can be achieved using multiple layers of blended textures and lightweight management of the texturing data is a challenge that has been addressed by previous practical methods ([6]). The texture coordinates for each mesh can be procedurally computed in shaders or can be a part of mesh data controlled by application, allowing painting editing. A mixture of these basic approaches can also be applied, in which the application controls the basic texturing over terrain, and the procedural rules, such as terrain slope restrictions, apply modifications on application provided data.

The implementation of this thesis follows a simple texturing approach with procedural texture coordinates and supports blending of three types of textures for the complete terrain. The textures that are applied to heightfield fragments are:

- 1D isohypse-like texture, that is used to convey height information through simple sensible coloring (making it easy to mark high grounds as snowy-white and low grounds as sand or grass),
- 2D pre-computed Perlin noise texture, that is used to introduce a high-resolution variation on the heightfield aiming to produce a simple grass-like effect, and
- 2D rock texture, that is used to paint the cliff-like steep regions on the heightfield.

The 2D textures are tiled on the heightfield using XZ world-space coordinates of height samples. Additionally, rock texture uses the slope (normal) and height information of the given sample height point as a variable blending factor parameter. The 1D height texture is directly mapped from the world-space height of the input point. Computing the procedural sampling points is done by low-cost shader functions, so have negligible effect on the overall performance for rendering. No transfer of per-vertex texture coordinates as part of mesh data is done.

The heightfield and the scene objects interacting with heightfield are lit using a sun light source, with constant light ray direction over the scene. The implementation includes no terrain self-shadowing or shadowing on terrain as caused by scene objects.

The lighting calculations are applied per-vertex if an adaptive renderer (except the simple variant) is not used. If it were applied per-fragment, the normal texture would be sampled per-fragment and visual pop-ups would appear since the normal texture resolution and so its content can be updated during the simulation. Per-vertex sample coordinates on normal textures are kept constant during the simulation, even if different resolution normal maps are sampled. Asserting that the normals on sample coordinates remain constant on a transition between low and high resolution texture of a patch is possible as shown in the implementation of this thesis.

Adaptive renderers (except the simple variant) re-calculate shading on deformed cells by using per-fragment height deformation look-up in fragment shaders, so they apply per-fragment lighting to deformed cells. Section 6.2.4.5 describes how regions lighted with per-pixel and per-fragment approaches are blended seamlessly. Refer to lighting-related processes in the flow charts presented in Figure 6.6 and Figure 6.6 to see when and how the lighting terms are set in the GPUshading programs.

6.2.3 Two-Step Sub-Patch Rendering for Deformed Patches

The first type of deformation shader renders deformed terrain patches in two steps. Figure A.1 shows the two passes applied on a terrain patch, along with wire-frame visualization of heightfield triangles. The first step renders non-deformed cells of deformed terrain patches. In this step, each patch is rendered in a single draw call and deformed cells of the patch are skipped (not triangulated) in the index data generation phase, producing gaps on heightfield surface. High-resolution normal map data is used as the height sample source, because of the requirement to keep the normals of the surface consistent between deformed and non-deformed patches. The second step renders the deformed cells, filling the gaps opened in the first phase, using high-resolution heightfield data as the height sample source. Each deformed cell in low-resolution grid is rendered in a single draw call, so filling in all the gaps of a patch require multiple draw calls, which use the same underlying base mesh.

Notice that sub-patches are rendered only when the patch holding the sub-patch is rendered in highest level-of-detail layer, where every low-resolution grid cell can be triangulated. Adaptive shading approaches does not suffer from this limitation, yet they can produce sharp and relatively distracting deformation rendering on deformed cells when the terrain patch is rendered with a low level-of-detail layer.

To speed up rendering, the non-deformed terrain parts are rendered as a batch and sub-patches in deformed patches are rendered in another batch, sub-patches of a patch being further grouped together. Since sub-patches use the same vertex data, the only render state difference between rendering different terrain patches are the high resolution heightfield data bindings and the patch/sample transformations.

A terrain sub-patch does not span the complete terrain patch region. A sub-patch's region is dependent on the position of the terrain patch over large-scale heightfield and the position of the sub-patch on the owner patch. To render sub-patches correctly, patch and sample transformations are handled differently.

The modification of the patch transformation data described in Section 6.2.1 is as the following. The patch scale factor is computed by using the equation $patchScale = 1.0/2 \times patchEdgeGridCount$ and it is shared by all sub-patches. The patch translation factor is dependent on $patchEdgeGridCount$, the row and column number of the patch cell that the sub-patch replaces. The x-translation is computed using the equation $translate[x] = 1.0/(patchEdgeGridCount \times 2) + patchCol/patchEdgeGridCount$. The y-translation is similar to x-translation. The only difference is that $patchRow$ is used instead of $patchCol$ value. The sample transformation, which transforms the vertex coordinate on unit square to the sample coordinate of heightfield textures, is dependent on the position of the sub-patch on the region of its owner terrain patch.

6.2.4 Single-Step Rendering for Deformed Patches

The methods described in this section render a deformed terrain patch in a single pass. High resolution deformed and non-deformed vertex displacement maps and the high resolution normal map, as stored in terrain patches, are used as heightfield geometry sources. Figure A.3 shows non-textured and lighted visualization of a deformed heightfield using the two-step rendering and the single step rendering methods in this section.

The adaptive-type methods presented in this section are based on detection of deformed terrain cells in the fragment shader of the visualization pipeline, thus avoiding to render such deformed cells in a separate pass, as has been described in Section 6.2.3. As shown in Figure 6.6, the adaptive shading methods sample high-resolution deformed and non-deformed heightfield textures and detect deformed cells by comparing the sample height values from the two textures. Then, normal mapping or parallax mapping based shading are applied to the deformed cells. The adaptive methods also require blending methods to avoid rendering artifacts within the heightfield block of the deformed patch. The related details are presented in this section.

6.2.4.1 Simple Shading

Single-step simple shading method for deformed patches is an outcome of the heightfield data resolution update when rendering deformed regions. This shading method does not detect heightfield cells that have been deformed, so it follows a simpler path in the fragment shader without any additional heightfield texture lookup, thus increasing render speed significantly. The deformations can be only detected in the vertex shader and thus, only the deformations in the samples shared between high and low resolution heightfield data can be visualized with this approach. Furthermore, as the active LOD layer of a deformed patch decreases, the patch will be tessellated less, and the deformations are less likely to be captured. Since it does not process many of the high-resolution deformed heightfield samples, this method provides a crude approximation of the underlying deformed geometry. Yet, this method can visualize the deformations on terrain in moderate distances in a cost efficient manner, especially when merged with heightfield deformation enhancements calculated using per-vertex deformation amounts (see Section 6.2.5).

6.2.4.2 Adaptive Normal Mapping

Single-step adaptive normal mapping is the first adaptive shading approach for deformed patches. It is based on sampling high resolution normal maps in fragment shading stage and calculating lighting over surface in the fragment processing stage of rendering pipeline using the normal values sampled in high-resolution. This method provides an enhancement over the simple shading method, because over a low-resolution triangle surface, the normal values and so the lighting are calculated in higher frequency, producing a sharper rendering of the deformation, as seen on Figure A.3c.

6.2.4.3 Adaptive Parallax Mapping

While there exists numerous approaches for parallax mapping, the current methods are not directly applicable to the proposed heightfield system because of the

following facts:

- The available methods assume that the detail-surface is planar. In the current system, the surface itself is also a relatively dense triangulated heightfield.
- The available methods assume that the view ray direction is directed inside the detail-surface. However, in our system, the viewer can be below the surface looking upwards and view rays can point outwards of the plane and still intersect with height cells above.
- The available methods work on a generalization over arbitrary surface directions. The model space of the block to be parallax-mapped is in the same orientation of the world-space. Such conversions are not necessary. Thus, the requirement to calculate surface tangent and bi-tangent vectors per vertex is avoided.
- The parallax factor is computed only in deformed regions, not over the whole surface.

The followed parallax mapping method is based on [45]. Using the view ray into the vertex, the obliqueness of the view angle and a constant parallax scale (multiplier) factor over the surface, a per-vertex parallax vertex factor is computed. This factor is available to fragment shader in interpolated form inside the surface triangles, and it is used as the ray that is traversed on the processed pixel on the surface. The frequency of the linear sampling along the ray is an externally controllable parameter that affects the computation speed and output quality of the shader. After an intersection with the heightfield is found, the final parallax factor at intersection point is computed using a linear interpolation of the values obtained in the last two steps. The world-space of the intersection point is obtained using the world-space coordinate of surface point and the world-space parallax factor. The texturing and normal map sample point is updated, affecting the final pixel output.

The advantage of parallax mapping over normal mapping is that it can detect sharper features on the deformations on heightfield, as shown in Figure A.2.

Sub-patch rendering is used as a reference correct visualization of he high resolution deformations, since the triangulation of the surface follows the deformed data exactly. The parallax effect can be better observed and compared to normal mapping approach when the camera moves interactively over the deformed surface.

6.2.4.4 Additional Discussions on Single-Step Renderers

One minor disadvantage of single-step renderers, which are triangulated over low-resolution data is that the high-frequency details cannot be captured on the depth buffer. Such a case is shown in Figure A.10. The deforming object is shorter than the maximum amount of compression and low-resolution triangulation can pass through object geometry. To alleviate the this problem, the depth of fragments processed in deformed regions can be modified. The normal-map renderer can modify the depth based on the deformation amount, while the parallax renderer can modify the depth by projecting the world-space coordinate of the intersection point with the parallax ray and heightfield. Sample images demonstrating this approach is shown in Figure A.11. While the depth values generated are not perfect, most of the object can be uncovered. Further development and analysis can be made on how to set the depth buffer correctly to reduce the artifacts as much as possible.

6.2.4.5 Adaptive Shading Deformed and Undeformed Cell Blending

Deformed heightfield patches store both deformed and undeformed high resolution height sample data. Deformed regions and undeformed regions can be shaded using different execution paths in the shaders as discussed above, so their outputs may vary. A difference between regions shaded differently produce visible artifacts along the edges of deformed and non-deformed cells. To avoid these artifacts, blending regions between different shaded cells can be set up. In these blending regions, both the shading results must be calculated and be smoothly interpolated.

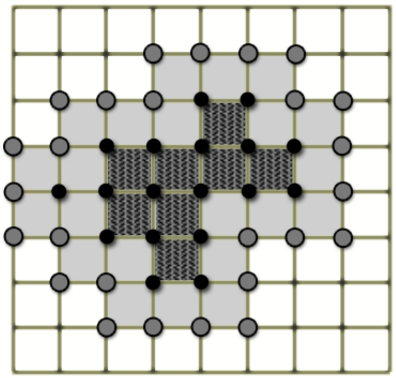


Figure 6.7: A deformed terrain patch with deformed and affected samples.

Figure 6.7 shows a sample 9×9 heightfield patch. The black dots denote samples that have been deformed. The blending regions are shaded in light gray. The region completely in between the deformed samples, as shown in a dark pattern, can be applied deformation shading methods without any blending requirements. However, since height values are interpolated inside the cells, the regions filled by the lighter color are also affected from the deformations of height samples and they also need to be affected from deformed region rendering methods, if any. The outer lighter height sample dots present the limits of such affected regions on heightfield. Notice that the affected cells are the cells which have at least one deformed sample in a corner, but not all of their corners are deformed. The transition region between different shading regions can be the deformation affected cells or the cells that are in the 8-neighborhood of these affected cells. The detection of affected cells is easier since the cell corner height values are available in the shader and the available height values also allow querying whether a sample is deformed or not. If 8-neighborhoods of affected cells are used as the blending region, the number of height samples that needs to be taken around a cell increase, thus reducing the performance of the shader. Thus, the light gray regions in this figure denotes both the deformation affected cells and the rendering transition regions, as used in the implementation.

6.2.5 Deformation Shading Enhancements

In addition to visualizing surface geometry (position and normal) updates in deformed regions, one can also visualize terrain deformations using terrain render

material updates. For example, such rendering updates may aim to mimic muddy ground on deformed cells by blending a mud-like color. The rendering material properties that can be updated span a wide range of lighting parameters such as diffuse color (either static or from a texture sample) or specular term multipliers.

The deformation shading enhancements use deformation blend factors that can either be set in vertex shader or fragment shader. The deformation shading methods that does not apply per-fragment cell deformation detection can use blending factors as computed in vertex shader, while other, adaptive, methods can also generate blending factor in fragment shader. Per-vertex blending factors are interpolated inside the triangulation over heightfield surface and since the triangulation is dependant on low-resolution level-of-detail layer of a patch, such values may not generate correct high-resolution blending factors. In return for simpler approximations of blending factors, per-vertex blending factors for rendering enhancement offer increased speed.

Figure A.4 shows the effects of a simple deformation shading enhancement, which blends a brown color as the terrain compression increases. Notice that such shading enhancements can greatly increase perception of deformation by relatively simple and computationally cheap models. Thus, it can be advised to introduce such enhancements when possible while designing a deformation system for specific terrain render materials.

6.2.6 Level-Of-Detail for Deformed Cell Shading

The rendering methods for deformed patches span a wide range of features, visual quality and computational cost. To further optimize the rendering pipeline, a per-patch level-of-detail system is designed. A final analysis of the methods developed fore deformation rendering methods is presented in Table 6.2.

The following additional observations are helpful in identifying a LOD system for deformation rendering:

- The deformations in long distances should also visible, yet high quality rendering is not required.

Table 6.2: Comparison of deformation rendering methods

Method	**Lighting**	**High-Res Sampling**	**LOD Dependent**	**High-Res. Tessel.**	**Steps**
Non-Deformed	Per-Vert.	No	-	-	1
Sub-patch	Per-Vert.	Vert	Yes	Yes	2
Simple	Per-Vert.	Vert	Yes ([1])	No	1
Normal-Map	Per-Frag.	Vert+Frag	No	No	1
Parallax-Map	Per-Frag.	Vert+Frag	No	No	1

- If a patch is deformed, we can render the patch in a higher level-of-detail if it helps to make deformations visible and blending smoother.
- If boundary cells are deformed, neighboring patches should be rendered with the same method, to prevent visual artifacts (such as height cracks or surface lighting discontinuities) on patch boundaries.
- Single-step simple shading depends on the triangulation of heightfield patch surface, so low triangulation amount may not be able to catch per-vertex deformations.

Since deformation enhancements are a cheap and efficient way of marking deformed regions, it is assumed that this approach is active for all methods and deformed patches. The hardware capability of the machine can also be used to adjust the level-of-detail approach of the system. For example, if per-pixel processing has a high cost in a current configuration, adaptive methods may not be used to render any of the deformed patches. The presented approach assumes that the hardware is capable of running all the methods without any significant drawbacks.

In turn, the developed level-of-detail approach is as the following:

- The -sorted- deformed patches are evaluated with respect to their distance to viewpoint and categorized into four groups: Far, moderate, close and very close distance. Very close distances are assigned sub-patch renderer since it offers the highest quality rendering of deformed regions. The close

[1]The quality of result depends on the patch triangulation (LOD setting)

distances are assigned parallax renderer, moderate distances are assigned normal maps renderer and the far distances are assigned simple renderer.

- The neighboring patches which have deformations on their boundaries are updated to share the same deformation types and also preferably the same LOD layer. This adjustment rule takes advantage of the distance-sorted patch list to detect neighbors. An important detail is that patches that have deformed boundary cell should not use parallax renderer, which cannot traverse rays on neighboring patches. Such patches can use normal-map or sub-patch type renderer.

- Depending on the category, the LOD levels of the patches are adjusted, increasing active LOD layers so that it is more suitable for the selected deformation type. The LOD of the patches that will be rendered in sub-patch mode are set to highest, so that sub-patches can be generated. The LOD of the patches assigned simple renderer should have high LOD levels as denote previously, so LOD of such patches are set to highest level. The LOD levels of adaptive renderers need not to be updated, but minor increments can increase the sampling of low-resolution normals on the surface and create more detailed low-resolution renderings which can better blend visually with adaptive deformed region renderings.

- Finally, the patches are distributed to the render queue of the deformation renderer they are assigned to, which are then processed sequentially.

Also, another criteria for selecting the rendering method can be the amount of maximum(or average) compression inside a terrain patch. High compressions cannot be visualized with simple rendering or even normal-mapping, while sub-patch and parallax renderers can display sharper deformations in high-resolution data since their surface sampling resolution is higher than the other methods. However, the maximum compression amount of a patch is not available in the CPU in the current implementation and this approach is not implemented in the sample application.

Chapter 7

Implementation and Performance

This chapter presents additional implementation approaches that have been followed and the performance of the reference implementation of the proposed system. The CPU-side implementation of the sample application has been made using C++, compiled by Visual Studio 2005 and GPU-side simulation and visualization implementation is done using OpenGL API to control the GPU hardware. wxWidgets[1] library is used as the GUI framework of the application.

7.1 Scene Setup

The scene setup, as shown in the images Section A.4, consists of a size-adjustable heightfield, dynamic collidable scene objects and a sky-mapped background texture.

The low-resolution heightfield size is either the size of a loaded heightfield image or the size requested when heightfield generator is used. The tested heightfield sizes vary between $(2^{10}+1) \times (2^{10}+1)$ to $(2^{12}+1) \times (2^{12}+1)$. The patch and sub-patch sizes can be adjusted before setting the heightfield data.

The following types of objects, as also shown in Figure A.7 can be spawned (created) into the virtual world:

[1] `http://www.wxwidgets.org/` (LGPL License)

- Marble [sphere] (Figure A.7a): It is a perfect spherical UV-mapped sphere, textured with a rock texture.
- Crate [box/square cuboid] (Figure A.7b): It is a perfect rectangular prism with all faces of equal size.
- Table (Figure A.7c): The difference of this object to the other objects in the scene is that it holds multiple box-shaped physics collision geometric primitives attached to the single rigid body, so that the concavities in the object can be represented in object-object intersections.
- Torus (Figure A.7d): This mesh (ordinary torus) is a genus-one surface. Its moment of inertia and volume is computed using analytic forms, assuming that the mesh is filled inside. The density of this mesh is set to a high value, making the torus a heavy object.
- Stanford Bunny (Figure A.7e): The bunny is the only object setup that includes a heightfield collision mesh proxy with reduced vertex count.

To render background environment, a sky map is used. The sky-map texture shows a mountain region on a partly cloudy day and it matches with terrain textures and the lighting model that has been used. The fragments are directly colored from the source cube texture and no additional lighting calculation are applied to the sky map.

The Stanford Bunny model is freely available for non-commercial use at The Stanford 3D Scanning Repository[2]. The chair model is created by Jan Thorsen and distributed free for non-commercial use[3]. Crate texture is downloaded from TurboSquid[4]. The rock texture is also freely available[5]. The artist of the sky-map texture is Hazel Whorley, and the textures are available under Creative Commons Attribution-Non-commercial 3.0 license[6].

[2] `http://graphics.stanford.edu/data/3Dscanrep/`
[3] `http://www.3dville.com`
[4] `http://www.turbosquid.com/3d-models/free-x-mode-crates/348777`
[5] `http://www.seamlesstextures.org`
[6] `http://www.hazelwhorley.com/textures.html`

7.1.1 Generating Procedural Terrains

The sample application supports procedural generation of large-scale terrains. The images captured from the implementation use terrain models generated using the procedural methods described in this section. The procedural terrain generator GUI is shown in Figure 7.1.

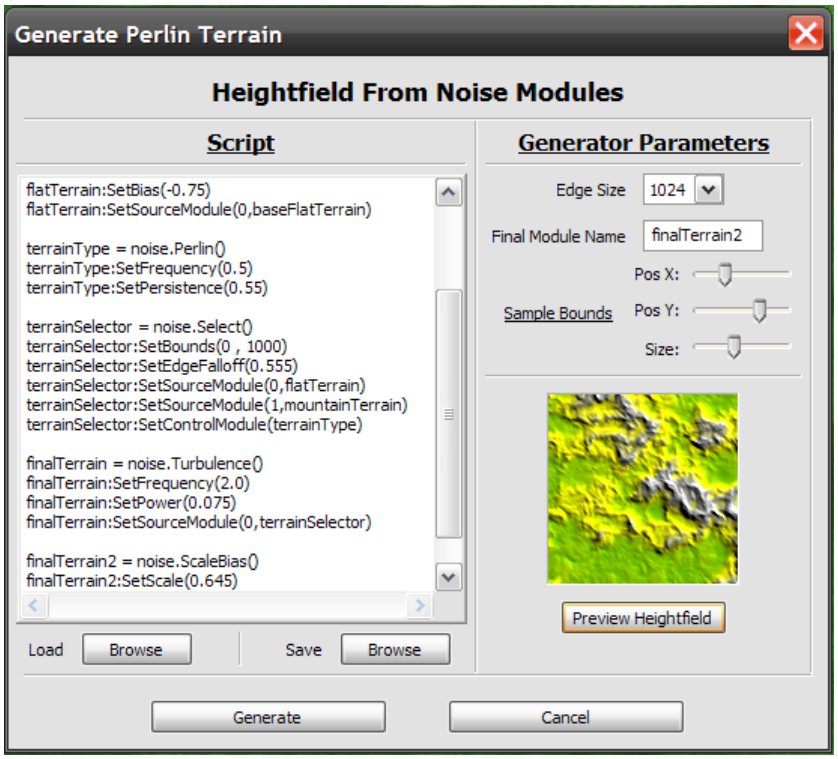


Figure 7.1: Procedural Terrain Generation User Interface

The generator models are based on Perlin noise model [37]. Libnoise library[7] is used as the noise generator software component. The Perlin-noise based modelling cannot natively simulate erosion effects on terrain, but can be used to generate highly realistic heightfields if multiple noise modules are merged to generate a final heightfield. Terrain is generated from text script files that can easily be modified, saved and loaded by the user. A scripting interface for libnoise is implemented using Lua scripting language[8]. The implementation uses luabind[9] library to bind the C++ interface of libnoise to the C interface of Lua. Finally, the previews of the final terrains are shown as a small texture on the user interface to assist the

[7] `http://libnoise.sourceforge.net/` (LGPL license)
[8] `http://www.lua.org/` (MIT license)
[9] `http://www.rasterbar.com/products/luabind.html` (MIT license)

user when modifying terrain script text.

7.1.2 Rendering Engine Implementation

The rendering engine implementation used in this thesis is Open Rendering Engine (OpenREng)[10], which has been partly developed in the course of the implementation of this thesis. OpenREng is an open-source rendering engine that uses OpenGL ES 2.0 and OpenGL 3.0 and above API's as the underlying graphics API, providing support for both mobile and desktop platforms. This software library does not use the deprecated GPU graphics specifications and by being driven by the latest shader pipelines and GPU buffers, it supports fast next generation and high performance rendering techniques. Currently, OpenREng aims to provide easy manipulation and definition of 3D scenes and additional architectural support for advanced rendering techniques. The components of the rendering engine include OpenGL wrappers, material system, meshes and external mesh loading, scene graph, camera, lighting, geometric primitives and render queues.

7.2 Performance

This section first discusses the testing approach and configurations, then presents an overview system performance followed by more detailed performance tests.

For testing time performance, high-resolution performance counters on CPU's (with granularity up to micro-seconds) and high-resolution GPU timer queries are used. The GPU timer queries[11] asynchronously record the time passed for execution of a list of OpenGL commands, and report time ranges in nanoseconds, although the granularity of the GPU timers are hardware dependent.

In the following sections, only time performances are presented. Memory requirements for permanent data structures are straight-forward from the height-field sizes and heightfield data components and types. Additional memory used

[10] `http://openreng.sourceforge.com` (Apache License V2.0)
[11] `http://www.opengl.org/registry/specs/ARB/timer_query.txt`

by internal steps are temporary and insignificant. To briefly denote heightfield memory requirements, a terrain patch uses 4 bytes for a low-resolution cell (1 texture holding 1-component 16bit data and 1 texture holding 2-component 8bit data) and 8-bytes for high-resolution cell (3 textures holding 1-component 16bit data and 1 texture holding 2-component 8bit data). So, a complete terrain of low-resolution size 2049×2049 with low-resolution patch size 33×33, high-resolution patch size 129×129 and 20 patches deformed uses around 17.5MB (for low-res) and 2.6MB (for high-res) of GPU memory to store per-cell heightfield topology information. The higher size of high-resolution data is compensated by the fact that there are fewer number of high-resolution data generated on the heightfield.

Please note that although many efficiency related improvements have been made in the sample application, it is not in a production-ready state and further optimizations in many stages are still highly possible. The implementation does not use hardware-specific extensions or routines and only depends on the core functionalities of latest OpenGL specifications. Likewise, multi-threading in CPU has not been used. The sample application developed is not a complete virtual environment application, and CPU is not used for any other main purpose other than rendering management and physical simulation, such as artificial intelligence. Some of the OpenGL commands are blocking, that is, the CPU needs to wait until all previous OpenGL commands are executed. It is more favorable if the CPU is kept busy before calling synchronization commands, but currently, since the number of tasks done by CPU is low, it stalls for some amount of time. So, more CPU tasks can be set up in the proposed system while the frame rate is not affected.

Unless otherwise is noted, the presented timings next are not averaged values of multiple frames, but values obtained from a typical single frame where the processing time is observed to be stable between the near frames.

7.2.1 Performance Overview

Table 7.1 shows the configurations of the test PCs used. The hardware spans mobile and desktop computers with varying CPU and GPU clock speeds. PC 1 stands as the fastest configurations in the available PCs, while PC 3 targets an

older PC that can support the presented system, offering real-time performance when the colliding objects are few. Please notice that the fastest hardware is not one of the fastest available hardwares at the time of writing of this thesis, so the results presented do not imply a performance limit of the proposed system. Most of the tests below are performed on the highest (PC 1) and the lowest (PC3) ends of available hardware. The same binary is tested on all platforms and the application is build as a 32-bit application.

Table 7.1: Test PC configurations

PC #	**GPU Model**	**Shader Clock Speed**	**VRAM**
1 (Desktop)	NVIDIA GeForce 9600 GT	1500 MHz	1024MB GDDR3
2 (Desktop)	NVIDIA GeForce 8800 GT	1500 MHz	512MB DDR3
3 (Laptop)	NVIDIA GeForce 8600M GS	900 MHz	256MB DDR2

PC #	**CPU Clock Speed**	**CPU Cores**	**RAM**	**Op. Sys.**
1 (Desktop)	2.67Ghz	4	4GB DDR3	Windows 7 (64bit)
2 (Desktop)	2.4Ghz	4	4GB DDR2	Windows 7 (64bit)
3 (Laptop)	1.8GHz	2	2GB DDR2	Windows XP (32bit)

The overview of timing performance of the proposed system is shown in Table 7.2. The camera is kept stationary and the objects are dynamically simulated for about 10 seconds. The view-port sizes are 1248×822(PC1), 1668×968(PC2) and 1268×718 (PC3). Some of the deformed patches are not visible and some deformed cells are not assigned to collision buffers since an object may no longer be on top of these patches. The physical simulation time includes collision detection between scene objects through their geometric primitives. Heightfield simulation time includes decompression, narrow-phase collision detection and patch normal updating steps. Physical data setup time includes the transfer of collision data from GPU memory to CPU memory and further processing of the textures on the CPU to create contacts.

Given the time measurements in Table 7.2, the rendering tasks, which also includes deformation rendering, takes the largest part of the frame time. Since the number of objects and generated contacts per object is relatively low in test cases, the physical simulation itself is observed to be fast. A fully enabled heightfield simulation with relatively low number of active collision buffers is shown

Table 7.2: Basic frame time performances (in ms) for a typical scene configuration

PC	FPS	Total Time	HF Sim.	Rendering	Phy Data Setup	Phy. Sim.
1	86.5	11.5	3.64	4.26	3.29	0.31
2	65.8	15.2	5.14	5.62	4.07	0.37
3	26.6	37.5	11.19	18.97	5.44	1.90

(a) The Scene Configuration And Additional Statistics

Terrain Size	1025×1025
Patch Low-Res Size	33×33
Patch High-Res Size	129×129
Scene Object #	11
Object Triangle #	75249
Deformation Renderer	Sub-patch
Decompression Type	Exp. Speed

Screen-Space Pixel Error	4
Visible Patch # (Undef)	276
Visible Patch # (Def)	10
Active Collision Buffer #	12
Deformed Patch #	16
Deform. Render Enhance.	✓
Collision Kernel Culling	✓

to take less time than the rendering task. The inclusion of heightfield simulation and rigid body collision detection with further physical simulation can be expected to reduce the frame rate to half, while the exact ratio depends on many parameters within the scene. More detailed analysis of the steps are presented in the following sections, including characteristics with respect to possible scene and terrain configurations.

7.2.2 Heightfield Collision Detection and Simulation Performance

Table 7.3 shows the overview performance of separate pipeline stages for the proposed heightfield simulation system. The large-scale heightfield size can only effect broad-phase collision detection phase, which is shown to take very little of the total simulation time. Similarly, among the terrain patch resolutions, only the high-resolution size is significant, since it also constitutes as the size of collision frame buffers. Collision buffer sizes noted in the tables are the same as high-resolution terrain patch resolutions.

Notice that the slowest stage of the heightfield simulation pipeline is the time required to update normals of updated and neighboring patches. Therefore, an optimization in normal map generator shader should be studied before the other

Table 7.3: Heightfield simulation performance overview (in ms)

PC	**Decomp.** *(GPU)*	**Broad-Phase CD** *(CPU)*	**Narrow-Phase CD** *(GPU)*		**Normal Upd.** *(GPU)*
			Obj. Data Gen.	Main Kernel	
1	0.67	0.10	1.70	1.13	1.96
2	1.10	0.17	2.28	3.5	2.78
3	1.78	0.32	2.25	2.97	7.15

Col. Buffer Size	129×129
Decomp. Model	Erosion
Object Count	46
Triangle Count	318027

Active Col. Buffer #	25
Normal Upd. LR Patch #	44
Normal Upd. HR Patch #	25
Collision Kernel Culling	✓

(a) The Scene Configuration And Additional Statistics

stages in the presented table. This behaviour has two explanations. The initial cause is the higher complexity and texture sampling requirements of the robust normal generator than the other GPU kernels used in heightfield simulation. The second cause is the requirement to update the neighboring patch normals when normals in the main patch is updated. Current sample application may be further optimized to check if deformations in bordering cells of the high resolution heightfield and signal normal update updates to neighboring patches accordingly, which can reduce the number of patches that are processed in normal update step. Since most of the collision pipeline is performed on the GPU (and controlled asynchronously by the CPU), the capability of GPU hardware affects the total execution time. Lastly, in the given scene configuration, PC 1 stands as roughly two times faster than PC 3, which stands as the lowest configuration available.

Table 7.4 further shows the time performance results obtained with differing collision buffer configurations, which are buffer size and the number of buffers active on a simulation step. In cases where the same number of cells are processed with different collision buffer resolutions and counts, the larger sized buffer configuration is processed faster, likely because of the fact that the number of frame buffer switches are reduced. Also, as expected, an increase in active collision buffer count affects the computation amount mostly linearly, while its effect on decompression kernel shows a slower gain. Since updates on terrain patch vertex displacement maps require an update in self and neighboring heightfield normals, the normal update kernel performance, which is performed per-frame after deformations, is also shown in this table, along with the number of high and low

resolution normal maps that have been updated. Note that the number of normal updates on high-resolution maps generally follow the number of active collision buffers and additional neighboring normal maps (high or low resolution) are also updated in this step.

Table 7.4: Heightfield GPU simulation performance (in ms) wrt. collision buffer configurations

Col. Buffer Size	**Active Col. Buf. #**	**Decomp. Kernel**	**Col.&Comp. Kernel**	**Normal Upd. Kernel**
65×65	4	0.19	0.52	0.77 / (7LR-5HR)
65×65	8	0.28	1.01	1.10 / (13LR-11HR)
65×65	16	0.42	2.05	1.51 / (21LR-20HR)
129×129	1	0.15	0.20	0.50 / (4LR-1HR)
129×129	4	0.23	0.51	0.88 / (8LR-4HR)
129×129	8	0.30	0.94	1.32 / (16LR-8HR)
129×129	16	0.40	1.20	2.07 / (30LR-16HR)
257×257	1	0.16	0.18	0.78 / (4LR-1HR)
257×257	16	0.65	1.46	3.54 / (31LR-16HR)
513×513	1	0.26	0.25	1.17 / (4LR-1HR)
513×513	8	0.61	1.70	3.90 / (14LR-8HR)

PC	1

Decomp. Model	Exp. Speed
Collision Kernel Culling	Yes

(a) Additional Configuration

Table 7.5 shows the comparison of the execution speed of different decompression models. The size and number of collision frame buffers are kept high to better observe kernel execution speeds. Decompression models use similar simple ALU logics so the expected execution times do not differ much. However since erosion model uses the local slope of heightfield and this data is sampled from a high resolution normal texture, the memory access causes the erosion model to perform slightly slower than the other two models. The effect of PC configuration to GPU kernels are also presented in Tables 7.5 and 7.6. As expected, newer generation GPU's with higher clock speeds and increased cores can significantly decrease the execution time.

Table 7.6 focuses on collision and compression kernel performance and related setup. When occlusion culling is active and fragments are discarded in case of no intersections, a manual copying of a heightfield texture is required, and this time

Table 7.5: Heightfield decompression models performance

(a) PC 1

Decomp. Model	Time (ms)
Lin. Speed	0.65
Exp. Speed	0.65
Erosion	0.79

(b) PC 3

Decomp. Model	Time (ms)
Lin. Speed	1.95
Exp. Speed	1.81
Erosion	2.85

(c) Additional Configuration & Statistics

Col. Buffer Size	257×257

Active Col. Buffer Count	18

is reflected in the timings achieved. The performance of the GPU also affects the results, while the performance gap between faster and slower hardware is larger when the buffer size is increased.

Table 7.6: Heightfield collision and compression kernel performance ith respect to hardware

(a) Collision buffer size 65×65, count 18

Occ. Cull	PC	Time (ms)
✓	1	1.95
	2	2.60
	3	3.19
×	1	0.92
	2	1.22
	3	1.82

(b) Collision buffer size 257×257, count 12

Occ. Cull	PC	Time (ms)
✓	1	1.28
	2	1.72
	3	4.01
×	1	0.85
	2	0.98
	3	3.04

Table 7.7 presents heightfield object collision data generation time statistics observed in run-times. In this phase, an object may be rendered to multiple collision buffers more than once because it may intersect multiple patches and thus be assigned to multiple collision buffers. The data is processed by using collision buffers as the primary index, thus a collision buffer is only activated once, binding GPU frame buffer resources and setting projection matrices for the attached terrain patch, while multiple objects, holding multiple triangles are rendered afterwards. As the number of objects in the scene increase, the expected number of active collision buffer count and the rendered triangle count increase. This trend also reflects to the sample values presented in Table 7.7.

Table 7.7: Heightfield object collision data generation performance

				Additional Statistics	
PC	**Col. Buffer Size**	**Total Triangle Count**	**Time (ms)**	**Total Object Count**	**Active Col. Buffer Count**
1	129×129	8400	0.50	10	8
1	129×129	181695	1.05	20	12
1	129×129	358764	1.30	62	12
1	129×129	393396	2.14	66	21
1	257×257	8400	0.29	10	4
1	257×257	331884	1.49	36	20
3	129×129	8400	0.72	10	8
3	129×129	331884	3.66	36	20
3	129×129	451188	4.02	70	28
3	129×129	474294	4.18	71	28
3	257×257	8400	0.67	10	4
3	257×257	331884	3.03	36	20

7.2.3 Heightfield Visualization Performance

Heightfield visualization performance can be studied under two main stages: low-resolution undeformed heightfield rendering with frustum culling and LOD optimizations, and high-resolution deformed heightfield patch rendering with different types.

First, I want to present some background information related to the rendering of heightfield. Frustum culling and active LOD is updated dynamically only when camera moves. Index buffer data is only updated when LOD configuration of a patch changes, as described previously. The index buffer update time includes both the generation of new index data and search and assignment of available index buffers to buffer requests. To render low-resolution (undeformed) heightfield data, a single shading program is activated for the complete undeformed heightfield, processing only visible patches after their LOD is set. The updated GPU states for rendering each patch includes uniform states, bound vertex displacement / normal textures and bound index data buffer. The polygon mode is set to filled rasterization during the tests.

The frustum culling time is not affected from deformation state of a patch.

Index buffer update time checks the sub-patch data of deformed patches only if sub-patch deformation rendering type is active. LOD selection time is based on calculating distance to the viewpoint. Also, as noted in Section 6.2.6, deformed patches may offset LOD results for better visualization of deformed regions. However, since the effect of deformations on a patch to computation time of frustum culling, index buffer generation and LOD selection is negligible wrt. other phases of the pipeline, it is not further analyzed in more detail in this section.

Table 7.8 shows the results obtained for undeformed heightfield visualization performance. The values are averages from a fly-through on terrain surface, which takes about 10 seconds of real-time. The heightfield and the path of camera is the same for all the test runs. Index buffer updates occur highly infrequently (updates occur once in every 50-60 frames on average) since the camera moves relatively slowly and the frame coherence is high, the average values of a common and specific number of updates per frame are given in the table. The table shows that frustum culling, LOD update and index buffer updates take only a small amount of time per frame, while they allow optimizations and reduction in the rendering data that is processed by the GPU.

Table 7.8: Visualization performance (undeformed) overview (in ms)

PC	**Frustum Culling** *(CPU)*	**LOD Update** *(CPU)*	**Index Buf. Update** *(CPU)*	**Viewport Size**	**Total frame time** *(CPU+GPU)*
1	0.05	0.006	0.16 (5Patch)	1428×822	2.47
2	0.06	0.006	0.13 (5Patch)	1668×968	2.93
3	0.11	0.029	0.48 (5Patch)	1268×718	5.75

Terrain Size	2049 × 2049
Patch Size	65 × 65

Screen-Space Pixel Error	6
Average Visible Patch #	85.9
Average Patch LOD	1.84-2.67

(a) The Scene Configuration And Additional Statistics

The effect of terrain rendering configuration (terrain size, patch size and visible pixel error in screen-space) to the low-resolution, undeformed patch render speed is presented in Table 7.9. The performance is given in frames-per-second (FPS) rather than milliseconds, which is a more common approach to presenting rendering performances. The scene and camera are static, so no frustum or LOD

updates are done during the frame. LOD levels are set so that 0 maps to the highest resolution geo-mipmap of a patch, while patch size determines the maximum supported LOD layer of a patch. In the columns denoting FPS values with exact LOD layer denoted, all visible patch active LOD layers are set to the value in the column header. The machine used for this setup is PC 1, and the screen view-port size is 1240×850 for all the test runs.

As seen in Table 7.9, the overall frame-rate in all tested terrain configurations, with the dynamic LOD setting with 4 pixel errors allowed, is between 270 and 190 FPS. For all tested terrain sizes, patch size of 65×65 generates the fastest frame rate. It can be claimed that since the number of vertices inside the view frustum are the same among different patch sizes for similar scene and camera configurations, the total FPS values observed are similar. The visible number of patches do not differ much between terrains of size 2049×2049 and 4097×4097, because the whole terrain becomes larger than viewing frustum and the remaining patch blocks in larger heightfield are not shaded by the GPU. Under visible patches column, notice that frustum culling can significantly reduce the number of drawn patches. Total patch count in a complete heightfield is $((TerrainSize - 1)/(PatchSize - 1))^2$.

Table 7.9: Visualization FPS performance (undeformed) wrt. terrain render config.

Edge Size		**Visible**	**Level-Of-Detail**								
Total	Patch	**Patch #**	0	1	2	3	4	5	6	7	Dynamic (4px err)
1025	33	617	132	219	280	320	335	346	-	-	**252** (Avg. LOD=3.35)
1025	65	175	138	247	325	368	394	425	439	-	**259** (Avg. LOD=2.83)
2049	33	1021	101	172	228	252	252	252	-	-	**230** (Avg. LOD=3.56)
2049	65	275	107	217	307	355	396	400	398	-	**276** (Avg. LOD=2.96)
2049	129	78	100	211	310	360	390	405	390	415	**248** (Avg. LOD=2.25)
4097	33	1016	95	154	192	210	211	211	-	-	**192** (Avg. LOD=3.94)
4097	65	286	101	212	298	345	378	398	408	-	**274** (Avg. LOD=3.53)
4097	129	85	92	196	285	332	356	385	390	390	**243** (Avg. LOD=2.91)

To compare shading time performances of the presented deformation methods, Table 7.10 is presented. Low-end and high-end PC configurations are tested with only a single patch deformation. Only the deformed patch is rendered during the tests and the GPU time required to draw the deformed patch is queried. No decompression is applied to the deformed cells during the tests. In the test

scene, about one third of the patch surface is deformed, thus the high-resolution triangulation covers a large area and requires many sub-patches to be rendered. The ratio of deformed surface to complete surface affects all the renderer methods except the simple renderer. In close-view test case, the patch covers most of the viewport of the application. About 1/16th of this region is covered in the far-view test case. Single-pass adaptive renderers are shown to be the slowest renderers in the near-view test cases, since they use per-fragment sampling of normal values and apply per-fragment lighting. Parallax mapping method also slows down the renderer significantly. The overhead of sub-patch methods with respect to the simple method is that it draws the internal low-resolution cells one by one. The characteristics of rendering performances between close and far views differ as expected. Normal-mapping in far-view generates outputs faster than sub-patch method. The simple method also runs faster in far-views, because the LOD layer of the patch is decreased and surface is more sparsely triangulated. Please note that the sub-patch renderer has been selected as a reference renderer which produces high quality triangulations over the surface and the projected screen-space size of the patch affects the rendering speed of adaptive methods.

Table 7.10: Visualization performance (deformed) overview

Shading Model	**Time (ms)**
1P Sub-Patch	4.17
2P Simple	1.57
2P Normal	6.78
2P Parallax	19.73

(a) PC 3 (close-view)

Shading Model	**Time (ms)**
1P Sub-Patch	1.27
2P Simple	0.44
2P Normal	1.88
2P Parallax	6.62

(b) PC 1 (close-view)

Shading Model	**Time (ms)**
1P Sub-Patch	1.16
2P Simple	0.09
2P Normal	0.37
2P Parallax	1.76

(c) PC 1 (far-view)

7.2.4 Rigid Body Simulation Performance

The rigid body simulation is the last procedure in a given simulation & rendering frame. The underlying physics engine (ODE) provides a single interface for stepping a simulation world, which both resolves the constraints and integrates the rigid bodies using the new angular and linear velocities. The presented physical simulation step timings in this section therefore include both solving contacts and updating bodies, but does not include inter-object collisions, which has been

disabled to measure the time required only to solve object-heightfield interactions.

To achieve increased physical simulation accuracy, ODE is configured to operate on double precision floating points. The scene has a default gravitational force applied to all rigid bodies within, driving the simulation forwards even when no external forces are applied. The bodies have both linear and angular velocity damping and auto-disabling features under small threshold values, increasing stability of simulation and helping the objects to rest gracefully. The bounciness parameter of contact points generated are set to zero.

In the current implementation, collisions are found once for each frame, while the same contact data is used for multiple iterations of physics stepper. An important point with respect to simulation stability is that the intersections are solved after they occur. In case a full rendering and simulation frame takes long to complete, the object intersection amounts into the heightfield are likely to increase, since the total stepping time is increased. Such deep intersections are harder to solve and can cause popping effects on the objects that have penetrated the heightfield. The time performances presented here are within the stability range of physical simulation for each PC configuration.

Table 7.11: Rigid Body Simulation Performance

PC	**Scene Body #**	**Avg. Contact # Per-Body**	**Stepping Time (ms)**	**Internal Step #**	**CPU Time (ms)**
1	11	0.97	16	4	0.20
1	11	10.2	15	3	0.39
1	35	6.48	30	6	1.03
1	40	6.82	39	8	2.90
3	11	8.91	33	7	1.16
3	11	4.13	31	6	0.34

Table 7.11 shows the basic physical simulation performance. The stepping time is the time that has passed since the last iteration of the physics world (thus it is the frame time). The given step time is divided into internal steps of constant time spans, which is set to 5ms for the current system. The constant step-time is an important factor for the stability of the integrator. Higher number of contacts assigned to a single rigid body increases the solution time of the system as expected. But, it has been observed that for relatively small sized objects as has been tested in the sample application, the number of contact data generated

per frame is low.

Also, another point worth mentioning is that the physical integrators speed performance is highly affected from contact geometries and surface parameters. Given constraints to the system may not lead to a convergence to solution in small number of steps (or even a solution may not be a possible). Surface parameters, such as friction amounts, friction models and error relaxation parameters, also can significantly affect the solution time of given constraints. Thus, the numbers presented here are sampled from stable frames and are given as a reference only. Due to the discussion above, they cannot denote upper or lower bounds for the time required to solve the physical simulation. Thus, in Table 7.11, although the observed parameters of the integrator are similar, the execution times may differ substantially. Generating easily solvable / consistent constraints to the system is the important point that affects the performance of this step.

Table 7.12: Collision Data Setup Time

PC	**Collision Buffer Size**	**Active Collision Buffer #**	**Data Transfer Request to PBO**	**Processing Collision Data**
1	129×129	4	0.66	6.55
1	257×257	1	0.15	6.50
3	129×129	4	0.78	1.17
3	257×257	1	0.22	1.10

Table 7.12 shows the timings observed for reading back collision data from GPU to CPU and further creating contact structures for physical simulation engine. While the data transfer request takes shorter to complete in faster hardware, the processing of the data slows down. Although I tried my best efforts, I could not find the exact source of this slow down. Such a behavior was not reflected in Table 7.2, which was based on a slightly different version of the application. The processing time is affected greatly even by a simple pointer-chasing operation on the newer hardware and the problem is not related to the main body of the loop exceeding hardware cache size. I suspect the problem is related to the fact that the application is build for 32bit architectures and tested on a 64bit operation system.

The collision data read-back step uses pixel buffer objects[12] to speed-up the

[12] `http://www.opengl.org/registry/specs/ARB/pixel_buffer_object.txt`

read-back time whenever allowed by hardware. This is possible because of the transfer of collision data can be made through DMA (Direct Memory Access) while the read-back to DMA memory can be asynchronous started by the CPU [15]. When the DMA memory transfer continues for multiple collision buffers asynchronously, the application can process the available read-back data to create new contact and sub-patch structures as required.

Reading GPU results back to CPU stands as a possibly time-consuming task in the system as the data size gets larger, although the read-back can be performed asynchronously giving the CPU more time to do other tasks in the meanwhile. This read-back is required to generate contact data, to detect if the per-patch heightfield is deformed in the current step and to generate sub-patches (if sub-patch rendering is active/required). This read-back can be prevented as the following: GPU occlusion queries fully support boolean collision detection and can be used to detect deformations, however, sub-grid data structures cannot be generated on the CPU using solely occlusion queries, thus sub-patch based rendering for deformations is not possible in such a case. For porting the simulation of rigid bodies to GPU (using parallel ports of algorithms), time integration and collision solving are two main tasks. Time integration of multiple rigid bodies is an embarrassingly parallel task, which can be easily ported to GPU. However, the collision solving phase is a complex task, which can be performed in multiple Jacobian solver loops, for example, as presented by Tasora et al. [44].

Chapter 8

Conclusions and Discussions

8.1 Conclusions

In this thesis, a novel heightfield management system that can support deformations on heightfield and physical simulation of rigid bodies on the heightfield surface have been presented. The system is targeted towards interactive applications and the performance results show that the collaboration of GPU and CPU in an end-consumer PC configuration can allow real-time dynamic heightfield-based simulations within virtual scenes.

The presented methods are expected to be future-proof to advances in clock and memory access speeds in the GPU units. The basic idea behind heightfield simulation and collision detection methods presented in this thesis is processing of heightfield cells independently from other cells in the system and using no or small local information around the cell, while object collision data generation requires a rasterization of the triangles of the objects. For both of the tasks, GPUs offer as an efficient architecture. Also, many GPU kernels can be implemented in other recent GP-GPU targeted languages such as NVIDIA's CUDA [1] and OpenCL [2].

Although the current hardware capabilities cannot support high number of

[1] `http://www.nvidia.com/object/cuda_home_new.html`
[2] `http://www.khronos.org/opencl/`

colliding objects distributed over a very large heightfield with deformation simulations, relatively few number of objects can be dynamically simulated with an additional 1-2 ms computation time. So, the presented method can be applied for characters / objects that can attract high visual attention and thus, they will have more plausible interactions with the environment. The remaining objects can be simulated over the heightfield using simpler methods, such as using ray-casting or simple geometric primitive bounding volumes for intersections between objects and heightfields.

Also, the presented system can be used as only a collision detector between heightfields and triangular objects, which can then be used to physically simulate the objects. By setting height compression limits to 0, the heightfield cannot be deformed, yet required physical contact data can be generated on the CPU. Since the patches then cannot be deformed, deformation rendering and de-compression steps will be skipped, lowering the amount of computations required per frame.

8.2 Future Work

The methods presented in this thesis do not include displacement and erosion models. Displacement models can be used to transfer the compressed terrain volumes to the neighboring regions of deformations, thus increasing the height of the heightfield samples. This approach allows preserving some of the compressed volume information. The erosion models are also based on the flow of terrain material, yet this time aiming to approach to stable configurations after compression and displacement occur. The difference between the erosion models and the decompression models described in this thesis is that the latter does not model material propagation. As noted previously, the initial low-resolution heightfield is assumed to be in stable configuration, and deformation algorithms try to exploit this assumption to create a plausible animation. However, once deformed, the stable configuration may be derived from the deformation data dynamically, using erosion algorithms. Supporting displacement and erosion interactions on heightfield deformations, enhancing the reality of the results, can be developed as an extension to the presented work.

Another extension can be made to terrain to object interaction. Current proposal only discusses the case where the terrain applies a physical constraint using hard contact data after it is fully deformed. The other case where the object touches the ground, yet not compresses it fully can be analysed. This case cannot be modeled as a hard contact constraint, and a spring-like constraint needs to be defined. Another major future addition can be the use of deformable objects instead of rigid bodies that interact with terrain and each other.

Reducing the number of contact points between an object and the terrain stands as another future work. Some of the contact constraints can be removed from the system without affecting the simulation by analysing the geometric properties of generated contact points, because the other constraints may cover the effect of the removed contacts, which are probably found on the inner collision region.

The current collision detection method is a discrete method and it detects collisions after they occur. If the objects move fast, the system may fail to detect collisions in between the steps. Also, the physics solver cannot deal with deep object penetrations as it can with smaller penetrations. So, continuous collision detection methods can be developed to be able to better simulate the objects and the terrain.

As noted in Section 5.2, the implementation uses the contact depth data along the heightfield up-axis, while it should be along the normal axis. The related update can be implemented and the physics integrator stability increase can be observed on high-slope areas. Another contact data fix can be the computation of the first friction direction in contacts between objects and heightfield. The heightfield is a static geometry, so this step will need to make use of the object velocity direction on the contact point.

GPU shading techniques of deformations are also open to further development, such as taking advantage of programmable tessellators on GPU hardware. Parallax shaders can be improved to generate more accurate heightfield intersections on cases where the camera looks from below. Also, cases where the high-resolution deformed heightfield data cannot be rendered accurately with adaptive renderers, as shown in Figure A.10, can be better handled by including further extensions

to the parallax renderer, tracing rays on the surface where a deformation has not occurred, but the surface triangulation does not follow the height values.

Lastly, the current terrain management system does not support paging, that is, it requires all of the heightfield data to be in memory (both CPU and GPU). The heightfield management system can be updated to support such features and new rendering features such as self-shadowing of terrain can be developed on the current system.

Appendix A

GPU Shaders and Additional Figures

A.1 Object Collision Data Generator OpenGL Program

```
in vec4 re_Position;
invariant gl_Position;
void main () { gl_Position = re_ModelViewProjectionMatrix * re_Position; }
```

Listing A.1: Object Collision Data Generator Program - Vertex Shader

```
// OUTPUT's
invariant out uint outHeight; // 16 bit
invariant out uint outID;     // 8 bit
invariant out float outMass;  // 16 bit
// UNIFORM'S
uniform int objectID;
uniform float objectMass;
// CONST's
const float unitY = 0.005;
const float heightMult = float(65535); // 256*256-1);
// CODE
void setHeight(uint height) { outHeight = height; }
void setID(uint id)         { outID = id;         }
void setMass(float m)       { outMass = m;        }
void main() {
 // Note: Since we use orthographic projection, z is linear.
 setHeight(uint(gl_FragCoord.z*heightMult-1/unitY));
 setID(uint(objectID));
```

```
19  setMass(objectMass);
}
```

Listing A.2: Object Collision Data Generator Program - Fragment Shader

A.2 Heightfield Normal Generator OpenGL Program

```
in vec4 vTexCoord;
2
invariant out vec2 theNormals;      // high/low-res texture target
4
uniform usampler2D terHeightTex;    // high/low-res texture source
6 uniform usampler2D terHeightTex_L; // high/low-res texture source
uniform usampler2D terHeightTex_R; // high/low-res texture source
8 uniform usampler2D terHeightTex_U; // high/low-res texture source
uniform usampler2D terHeightTex_D; // high/low-res texture source
10
uniform isampler2D terHeightTex_Diff;   // high-res texture source
12 uniform isampler2D terHeightTex_Diff_L; // high-res texture source
uniform isampler2D terHeightTex_Diff_R; // high-res texture source
14 uniform isampler2D terHeightTex_Diff_U; // high-res texture source
uniform isampler2D terHeightTex_Diff_D; // high-res texture source
16
uniform int highResHFSize;
18 uniform int scaleFactor;

20 const float unitXZ = 1.0f;
const float unitY = 0.005;
22 const ivec2 offsetU = ivec2(0,+1);
const ivec2 offsetD = ivec2(0,-1);
24 const ivec2 offsetL = ivec2(-1,0);
const ivec2 offsetR = ivec2(+1,0);
26
// The texture lengths
28 int textureLength, textureLength_L, textureLength_R,
    textureLength_D, textureLength_U;
30
bool hr  () { return textureLength  ==highResHFSize; }
32 bool hr_L() { return textureLength_L==highResHFSize; }
bool hr_R() { return textureLength_R==highResHFSize; }
34 bool hr_U() { return textureLength_U==highResHFSize; }
bool hr_D() { return textureLength_D==highResHFSize; }
36
vec3 normal_CentralDifference(in uint curHeight, in uint heights[4]){
38  vec3 vU1 = vec3(0,unitY*(int(heights[0]) - int(curHeight)),+unitXZ);
  vec3 vD1 = vec3(0,unitY*(int(heights[1]) - int(curHeight)),-unitXZ);
40  vec3 vL1 = vec3(-unitXZ,unitY*(int(heights[2]) - int(curHeight)),0);
```

```
 vec3 vR1 = vec3(+unitXZ,unitY*(int(heights[3]) - int(curHeight)),0);
42
 vec3 nU = cross(vU1,vec3(+1,0,0));
44 vec3 nD = cross(vD1,vec3(-1,0,0));
 vec3 nL = cross(vL1,vec3(0,0,+1));
46 vec3 nR = cross(vR1,vec3(0,0,-1));
 return normalize(nU+nD+nL+nR);
48 }

50 vec3 normal_DiscreteDif(in uint curHeight, in uint heights[4]){
 vec3 dif = vec3(
52  float(int(heights[2])-int(heights[3])),
  float(int(heights[1])-int(heights[0])),
54  400 ); // some value
 return normalize(dif);
56 }

58 // Returns the normalized normal at the given sample position on the heightfield
 vec3 getNormal_TexelFetch(in vec2 texSamplePos){
60 uint curHeight = textureLod(terHeightTex,texSamplePos,0).r;
 uint heights[4]; // 0=UP 1=DOWN 2=LEFT 3=RIGHT
62
 float multp = float(textureLength-1);
64 ivec2 texelCoord = ivec2(round(texSamplePos*vec2(multp,multp)));
 //   return debugTexelCoord(texelCoord);
66
 ivec4 difDist = ivec4(textureLength,textureLength,textureLength,textureLength);
68 { // NOTE: ORIGIN IS LEFT-DOWN CORNER

70  // UP
  if(texelCoord.y!=textureLength-1) {
72  heights[0] = texelFetchOffset(terHeightTex,texelCoord,0,offsetU).r;
  if(hr()) heights[0] = uint(int(heights[0]) +
74  texelFetchOffset(terHeightTex_Diff,texelCoord,0,offsetU).r);
  } else { // ON THE EDGE
76  ivec2 texSampleCoord = texelCoord;
  texSampleCoord.y = 1;
78  // if resolutions differ, need to update texSampleCoord.x as well..
  if(textureLength_D<textureLength) texSampleCoord.x /= int(4);
80  if(textureLength_D>textureLength) texSampleCoord.x *= int(4);
  heights[0] = texelFetch(terHeightTex_D,texSampleCoord,0).r;
82  if(hr_D()) heights[0] = uint(int(heights[0]) +
  texelFetch(terHeightTex_Diff_D,texSampleCoord,0).r);
84  difDist[0] = textureLength_D;
  }
86
  // DOWN
88  if(texelCoord.y!=0) {
  heights[1] = texelFetchOffset(terHeightTex,texelCoord,0,offsetD).r;
90  if(hr()) heights[1] = uint(int(heights[1]) +
  texelFetchOffset(terHeightTex_Diff,texelCoord,0,offsetD).r);
92  } else { // ON THE EDGE
  ivec2 texSampleCoord = texelCoord;
```

```
94  texSampleCoord.y = textureLength_U-2;
    // if resolutions differ, need to update texSampleCoord.x as well..
96  if(textureLength_U<textureLength) texSampleCoord.x /= int(4);
    if(textureLength_U>textureLength) texSampleCoord.x *= int(4);
98  heights[1] = texelFetch(terHeightTex_U,texSampleCoord,0).r;
    if(hr_U()) heights[1] = uint(int(heights[1]) +
100 texelFetch(terHeightTex_Diff_U,texSampleCoord,0).r);
    difDist[1] = textureLength_U;
102 }

104 // LEFT
    if(texelCoord.x!=-0) {
106 heights[2] = texelFetchOffset(terHeightTex,texelCoord,0,offsetL).r;
    if(hr()) heights[2] = uint(int(heights[2]) +
108 texelFetchOffset(terHeightTex_Diff,texelCoord,0,offsetL).r);
    } else { // ON THE EDGE
110 ivec2 texSampleCoord = texelCoord;
    texSampleCoord.x = textureLength_L-2;
112 // if resolutions differ, need to update texSampleCoord.x as well..
    if(textureLength_L<textureLength) texSampleCoord.y /= int(4);
114 if(textureLength_L>textureLength) texSampleCoord.y *= int(4);
    heights[2] = texelFetch(terHeightTex_L,texSampleCoord,0).r;
116 if(hr_L()) heights[2] = uint(int(heights[2]) +
    texelFetch(terHeightTex_Diff_L,texSampleCoord,0).r);
118 difDist[2] = textureLength_L;
    }
120
    // RIGHT
122 if(texelCoord.x!=textureLength-1) {
    heights[3] = texelFetchOffset(terHeightTex,texelCoord,0,offsetR).r;
124 if(hr()) heights[3] = uint(int(heights[3]) +
    texelFetchOffset(terHeightTex_Diff,texelCoord,0,offsetR).r);
126 } else { // ON THE EDGE
    ivec2 texSampleCoord = texelCoord;
128 texSampleCoord.x = 1;
    // if resolutions differ, need to update texSampleCoord.x as well..
130 if(textureLength_R<textureLength) texSampleCoord.y /= int(4);
    if(textureLength_R>textureLength) texSampleCoord.y *= int(4);
132 heights[3] = texelFetch(terHeightTex_R,texSampleCoord,0).r;
    if(hr_R()) heights[3] = uint(int(heights[3]) +
134 texelFetch(terHeightTex_Diff_R,texSampleCoord,0).r);
    difDist[3] = textureLength_R;
136 }
    }
138 { // Re-Scale samples to low-resolution
    uint scaleFactorU = uint(scaleFactor);
140 if(difDist[0]==highResHFSize){
    if(curHeight>heights[0]){
142 heights[0] = curHeight-(curHeight-heights[0])*scaleFactorU;
    } else {
144 heights[0] = curHeight+(heights[0]-curHeight)*scaleFactorU;
    }
146 }
```

```
  if(difDist[1]==highResHFSize){
148 if(curHeight>heights[1]){
    heights[1] = curHeight-(curHeight-heights[1])*scaleFactorU;
150 } else {
    heights[1] = curHeight+(heights[1]-curHeight)*scaleFactorU;
152 }
  }
154 if(difDist[2]==highResHFSize){
  if(curHeight>heights[2]){
156 heights[2] = curHeight-(curHeight-heights[2])*scaleFactorU;
  } else {
158 heights[2] = curHeight+(heights[2]-curHeight)*scaleFactorU;
  }
160 }
  if(difDist[3]==highResHFSize){
162 if(curHeight>heights[3]){
    heights[3] = curHeight-(curHeight-heights[3])*scaleFactorU;
164 } else {
    heights[3] = curHeight+(heights[3]-curHeight)*scaleFactorU;
166 }
  }
168 }

170 // return normal_DiscreteDif(curHeight,heights);
  return normal_CentralDifference(curHeight,heights);
172 }

174 void main (void) {
  textureLength   = textureSize(terHeightTex  ,0).x;
176 textureLength_L = textureSize(terHeightTex_L,0).x;
  textureLength_R = textureSize(terHeightTex_R,0).x;
178 textureLength_U = textureSize(terHeightTex_U,0).x;
  textureLength_D = textureSize(terHeightTex_D,0).x;
180
  vec3 n = getNormal_TexelFetch(vTexCoord.xy);
182 theNormals = n.xz*0.5+0.5; // from [-1,+1] range to [0,1] range
}
```

Listing A.3: Heightfield Normal Generator Program Fragment Shader

A.3 Heightfield Rendering OpenGL Program

```
#if defined(PARALLAX) || defined(NORMALMAP)
2 #define ADAPTIVE
#endif
4#if defined(ADAPTIVE) || defined(SUBPATCH) || defined(SIMPLE_DEFORM)
 #define DEFORMED
6#endif
```

```
// BASIC INPUT's
in vec2 re_Position;
// BASIC OUTPUT's
invariant gl_Position;
out vec4 vVertexDiffuse;
out vec4 vTexCoords;
// BASIC UNIFORM's
uniform sampler2D terNormalTex; // low / high resolution
uniform vec3 patchTransform;    // x: scale XZ , y-z : translate XZ
// BASIC CONST's
const uint  maxHeightSample = uint(256*256-1);
const float unitY = 0.005;
const float grassTexScale = 0.025;
const float heightTexScale = 0.0035;
const float compressionRenderScale = 0.2;
// BASIC GLOBAL's
uint  terHeight;
float terHeight_F;
vec4  vertexPosWS;
ivec2 heightSampleTexel;

vec3 vVertexNormalMS;
vec3 vVertexNormalVS;

#if defined(DEFORMED) || defined(SUBPATCH_PRE)
 uniform usampler2D terHeightTex_Undef; // high resolution undeformed
 uniform isampler2D terHeightTex_Diff;  // high resolution difference
 uniform  sampler2D terNormalTex_Undef; // low resolution normal (undeformed hf)
 uint terHeight_Undef;
 int  terHeight_Diff;
#else
 uniform usampler2D terHeightTex;       // low resolution
#endif
#if defined(SUBPATCH) || defined(SIMPLE_DEFORM)
 out float vCompression;
#endif
#if defined(SUBPATCH)
 uniform vec4 sampleTransform;
#else
 const vec4 sampleTransform = vec4(0.5,0.5,0.5,0.0);
#endif
#if defined(ADAPTIVE)
 // Note: The third component holds the interpolated height value
 noperspective out vec2 vHFSamplePos;
 noperspective out vec4 vPosWS;
 out vec4 vPosVS;
#else
 vec2 vHFSamplePos;
#endif
#if defined(PARALLAX)
 const float parallaxRange = 10;
 out vec3 vParallaxOffsetWS;
#endif
```

```
62 // from norm. FP 2-component normal value, generates correct 3D normal vector
   vec3 genNormalFromTexVal(vec2 texVal){
64 ␣vec2 normXZ = texVal*2.0-1.0;
   ␣float normY = sqrt(1.0-dot(normXZ,normXZ));
66 ␣return normalize(vec3(normXZ.x,normY,normXZ.y));
   }
68 // Returns the normalized normal at the given sample position on the heightfield
   vec3 getNormal_TexelFetch(){
70 ␣return genNormalFromTexVal(texelFetch(terNormalTex,heightSampleTexel,0).xy);
   }
72
   void setVertexPosition() {
74 ␣vertexPosWS.w = 1.0;
   ␣vertexPosWS.xz = re_Position.xy*patchTransform.x + patchTransform.yz;
76 ␣vertexPosWS.y = terHeight_F;
   ␣gl_Position = re_ModelViewProjectionMatrix * vertexPosWS;
78 }
   void setVertexTexCoords() {
80 ␣vTexCoords.xy = vertexPosWS.xz * grassTexScale;
   ␣vTexCoords.z  = vertexPosWS.y * heightTexScale;
82
   ␣// calculate steepness texture mix
84 ␣float hmax = float(maxHeightSample);
   ␣float htexMax = hmax * 0.5;
86 ␣float blendFac;
   ␣if(terHeight<=htexMax)
88 ␣␣blendFac = terHeight/htexMax;
   ␣else
90 ␣␣blendFac = 1.0 - (terHeight-htexMax)/(hmax*0.5);
   #if defined(DEFORMED) || defined(SUBPATCH_PRE)
92 ␣vec3 lowResNormal = genNormalFromTexVal(textureLod(
   ␣␣␣␣␣␣␣␣terNormalTex_Undef,vHFSamplePos,0).xy);
94 #else
   ␣vec3 lowResNormal = genNormalFromTexVal(textureLod(
96 ␣␣␣␣␣␣␣␣terNormalTex,vHFSamplePos,0).xy);
   #endif
98 ␣vTexCoords.w = blendFac*smoothstep(0.57,0.58,1.0-lowResNormal.y);
   }
100 void setVertexLighting() {
   ␣vec3 lightDirVS = normalize(mat3(re_ViewMatrix) * re_SunLights[0].direction);
102 ␣vVertexDiffuse = vec4(0.45,0.45,0.45,1.0) * 1.4 + re_SunLights[0].color * 1.1 *
   ␣␣␣␣␣␣␣max(0.0, dot(vVertexNormalVS, lightDirVS));
104 }

106 void main (void) {
   ␣vHFSamplePos = re_Position.xy*sampleTransform.x + sampleTransform.yz;
108
   #if defined(DEFORMED) || defined(SUBPATCH_PRE)
110 ␣float textureCellCount = float(textureSize(terHeightTex_Undef,0)-1);
   ␣heightSampleTexel =
112 ␣␣␣␣␣␣ivec2(round(vHFSamplePos*vec2(textureCellCount,textureCellCount)));
   ␣terHeight_Undef = texelFetch(terHeightTex_Undef, heightSampleTexel, 0).r;
```

```
114  terHeight_Diff   = texelFetch(terHeightTex_Diff , heightSampleTexel, 0).r;
     terHeight        = uint(int(terHeight_Undef) + terHeight_Diff);
116 #else
     float textureCellCount = float(textureSize(terHeightTex,0)-1);
118  heightSampleTexel = ivec2(
         round(vHFSamplePos*vec2(textureCellCount,textureCellCount)));
120  terHeight        = texelFetch(terHeightTex, heightSampleTexel, 0).r;
     #endif
122
     // set vertex position
124  terHeight_F = unitY*float(terHeight);
     setVertexPosition();
126
     // set vertex normal
128 #if defined(SUBPATCH)
     if(terHeight_Diff==0){
130    // we need to mix the normal if adjacent to a deformed cell
       // assume that we were given the vertex on (-1,-1)
132    vec2  baseCoordF = vec2(-1,-1)*sampleTransform.x + sampleTransform.yz;
       ivec2 baseCoord  =
134        ivec2(round(baseCoordF*vec2(textureCellCount,textureCellCount)));

136    int  spGridSize = int(sampleTransform.w);
       vec3 c1 = genNormalFromTexVal(
138        texelFetch(terNormalTex,baseCoord+ivec2(0,0,),0).xy);
       vec3 c2 = genNormalFromTexVal(
140        texelFetch(terNormalTex,baseCoord+ivec2(0,spGridSize),0).xy);
       vec3 c3 = genNormalFromTexVal(
142        texelFetch(terNormalTex,baseCoord+ivec2(spGridSize,0),0).xy);
       vec3 c4 = genNormalFromTexVal(
144        texelFetch(terNormalTex,baseCoord+ivec2(spGridSize,spGridSize),0).xy);

146    // bi-linear interpolation of normal value
       vec2 interpFactor = re_Position.xy*0.5 + 0.5;
148    vec3 c12 = mix(c1,c2,interpFactor.y);
       vec3 c34 = mix(c3,c4,interpFactor.y);
150
       vVertexNormalMS = normalize( mix(c12, c34, interpFactor.x) );
152    vVertexNormalVS = re_NormalMatrix*vVertexNormalMS;
     } else {
154 #endif
     vVertexNormalMS = getNormal_TexelFetch(); // (re_NormalMatrix*re_Normal);
156  vVertexNormalVS = re_NormalMatrix*vVertexNormalMS;
     #if defined(SUBPATCH)
158  }
     #endif
160
     // lighting and texture coordinates
162  setVertexLighting();
     setVertexTexCoords();
164
     #if defined(SUBPATCH) || defined(SIMPLE_DEFORM)
166  vCompression = unitY*(max(-terHeight_Diff,0)) * compressionRenderScale;
```

```
#endif
#if defined(ADAPTIVE)
 vPosVS = re_ViewMatrix*vertexPosWS;
 vPosWS = vertexPosWS;
#endif
#if defined(PARALLAX)
 // the ray from viewpoint to vertex, both coordinates are in world-space
 vec3 viewRayWS = vertexPosWS.xyz - re_ViewPosition;
 vec3 parallaxDirection = normalize(viewRayWS);
 vParallaxOffsetWS = parallaxDirection * parallaxRange;

 // Note: This is a heuristic approach.
 // The more oblique the angle is, the more the parallax is expected to shift
 float parallaxScale = (length(parallaxDirection.xz)/parallaxDirection.y);

 if(parallaxScale>1) vParallaxOffsetWS *= abs(parallaxScale);
 // if the vertex is in higher ground than the view position,
 //we need a larger offset limit
 if(vParallaxOffsetWS.y>0) vParallaxOffsetWS *= 2;
#endif
}
```

Listing A.4: Heightfield Rendering Program - Vertex Shader

```
// BASIC INPUT'S
in vec4 vVertexDiffuse;
in vec4 vTexCoords;
// BASIC OUTPUT'S
out vec4 FragColor;
// BASIC TEXTURE SAMPLERS
uniform sampler1D heightTex;
uniform sampler2D grassTex;
uniform sampler2D rockTex;
uniform sampler2D terNormalTex; // low/high resolution
// BASIC UNIFORM'S
uniform int staticClr;
// BASIC CONST's
const uint  maxHeightSample = uint(256*256-1);
const float unitY = 0.005;
const float unitXZ = 2.0;
const float grassTexScale  = 0.025;
const float heightTexScale = 0.0035;
const vec4  staticColor = vec4(1,0,0,1);
const vec4  deformColor = vec4(85/256.0, 40/256.0, 20/256.0,1);

// ********************************
// PIPELINE-SPECIFIC DATA DEFINITIONS

#if defined(DEFORMED)
 uniform int enhanceDeform;
#endif
#if defined(SUBPATCH) || defined(SIMPLE_DEFORM)
 // INPUT
```

```
 in float vCompression;
#endif
#if defined(ADAPTIVE)
 // INPUT
 in vec4 vPosVS;
 noperspective in vec2 vHFSamplePos;
 noperspective in vec4 vPosWS;
 // UNIFORMS
 uniform isampler2D terHeightTex_Diff;  // high resolution
 uniform usampler2D terHeightTex_Undef; // high resolution
 // CONST's
 const ivec2 corner0 = ivec2(0,0);
 const ivec2 corner1 = ivec2(0,1);
 const ivec2 corner2 = ivec2(1,0);
 const ivec2 corner3 = ivec2(1,1);
 const bvec4 nonDeformedCell = bvec4(false,false,false,false);
 const bvec4 allDeformedCell = bvec4(true,true,true,true);
 // z-reject
 uniform float camPitchAngle; // in radians
 // GLOBALS
 float patchTexSize;
 float patchTexSizeMinOne;
 float texelOffset_F;
 vec2  multpHF;
 vec2  texelSize_F;
 bool  processingUndeformed;

 uvec4 hCell_Undef_U , hCell_Def_U ;
 vec4  hCell_Undef_F , hCell_Def_F ;
 float height_UnDef_F, height_Def_F;

 ivec2 baseTexelFetchCoord;
 vec2  nearestSample;        // The nearest sample (floored) on the hf
 vec2  nearestSampleDist;    // The distance to nearest sample
#endif
#if defined(PARALLAX)
 // INPUT
 in vec3 vParallaxOffsetWS;
 // UNIFORMS
 uniform ivec2 parallaxMinMaxSamples; // Parallax shading quality
 uniform ivec3 patchSizeHR_LR_Scale;
 // GLOBAL's
 vec3 parallaxOffsetWS_Norm; // normalized vParallaxOffsetWS
 vec4 texCoords;
 vec4 intersectPointWS;
 vec2 dx;
 vec2 dy;
#endif

// *************************
// LIGHTING
vec4 perPixelLighting(in vec3 normalVS){
 vec3 lightDirVS = normalize(mat3(re_ViewMatrix) * re_SunLights[0].direction);
```

```
83  return vec4(0.45,0.45,0.45,1.0) * 1.4 + re_SunLights[0].color * 1.1 *
        max(0.0, dot(normalVS, lightDirVS));
85  }
    vec4 perVertexLighting(){
87  return vVertexDiffuse;
    }
89  // NOTE:
    // If you want per-pixel lighting in non-deformed regions
91  // return perPixelLighting(normalize(vVertexNormalVS));

93  // ***************************
    // BASIC METHODS
95
    vec3 getNormalMS_LinearSamp(in vec2 texSamplePos){
97  vec2 normXZ = textureLod(terNormalTex,texSamplePos,0).xy*2-1.0;
    float normY = sqrt(1.0-dot(normXZ,normXZ));
99  return normalize(vec3(normXZ.x,normY,normXZ.y));
    }
101 vec3 getNormalVS_LinearSamp(in vec2 texSamplePos){
    return re_NormalMatrix*getNormalMS_LinearSamp(texSamplePos);
103 }

105 vec4 getTextureColor(){
    #if defined(PARALLAX)
107 if(processingUndeformed) {
    #endif
109   // we are using the texture coordinates as generated by the vertex shader
      // no need for explicit gradients
111   vec4 toRet = texture(grassTex, vTexCoords.xy);
      toRet *= texture(heightTex, vTexCoords.z);
113   return mix(toRet,texture(rockTex,vTexCoords.xy),vTexCoords.w);
    #if defined(PARALLAX)
115 } else {
      // use texture coordinates computed using parallax offsets
117   vec4 toRet = textureGrad(grassTex, texCoords.xy, dx, dy);
      toRet *= textureGrad(heightTex, texCoords.z, dx.x, dy.x);
119   return mix(toRet,textureGrad(rockTex,texCoords.xy, dx, dy),texCoords.w);
    }
121 #endif
    }
123
    // ***************************
125 // SAMPLING HEIGHTFIELDS

127 #if defined(ADAPTIVE)
    // NOTE: THE METHODS IN THIS PART ARE REMOVED
129 // SINCE THEY TAKE A LOT OF SPACE
    #endif // ADAPTIVE
131
    #if defined(DEFORMED)
133 void applyDeformEnhancement(){
    float compressionBlendFactor;
135 #if defined(SUBPATCH) || defined(SIMPLE_DEFORM)
```

```
    compressionBlendFactor = vCompression;
137 #elif defined(ADAPTIVE)
    // calculate floating point height values and interpolate
139 compressionBlendFactor = (height_UnDef_F-height_Def_F)*0.2;
    #endif
141 FragColor = mix(FragColor,deformColor,compressionBlendFactor);
    }
143 #endif // DEFORMED

145 #if defined(PARALLAX)
    vec2 convertToTS(vec2 x){
147 return x*float(patchSizeHR_LR_Scale[2])/(unitXZ*patchSizeHR_LR_Scale[0]);
    }
149 vec3 calculateParallaxAmount() {
    vec2  hfSamplePos = vHFSamplePos;
151 float heightHF    = height_Def_F;
    float heightRay   = vPosWS.y;
153
    // Want more samples when y is smaller (angle is oblique)
155 int numOfSteps = int( mix(parallaxMinMaxSamples[1],parallaxMinMaxSamples[0],
    abs(parallaxOffsetWS_Norm.y)) );
157 float stepSize = 1.0/float(numOfSteps);

159 float heightOffsetPerStep  = stepSize*vParallaxOffsetWS.y;
    vec2  textureOffsetPerStep = stepSize*convertToTS(vParallaxOffsetWS.xz);
161 float oldHeightHF, oldHeightRay;
    int stepNo;
163 bool exitEarly = false;
    for(stepNo=0 ; stepNo<numOfSteps && !exitEarly ; ++stepNo) {
165 oldHeightHF  = heightHF;
    oldHeightRay = heightRay;
167 // update ray height
    heightRay += heightOffsetPerStep;
169 // update hf height
    hfSamplePos += textureOffsetPerStep;
171 vec2 pNearestSample = texelSize_F * floor(hfSamplePos/texelSize_F);
    heightHF = hfInterp_Def( ivec2(round(pNearestSample*multpHF)) ,
173 hfSamplePos-pNearestSample );
    if(heightHF>=heightRay) exitEarly = true;
175 }
    if(!exitEarly) {
177 // NO INTERSECTION BETWEEN HF AND RAY
    //discard;
179 return vec3(0,0,0);
    }
181
    float delta1 = abs(oldHeightRay - oldHeightHF);
183 float delta2 = abs(heightHF - heightRay);
    float inBetween = (delta2/(delta1+delta2));
185 return vParallaxOffsetWS.xyz * (stepSize*(stepNo-inBetween));
    }
187

    // returns the shading (dffuse lighting) using parallax occlusion mapping
```

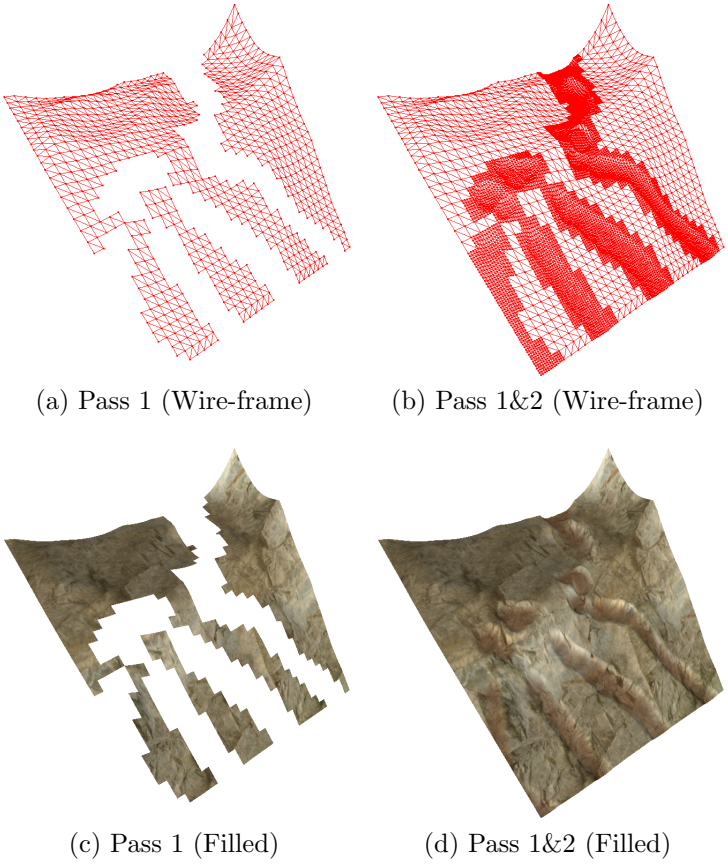
```
// updates texCoords variable
vec4 parallaxMap(){
 parallaxOffsetWS_Norm = normalize(vParallaxOffsetWS);
 vec3 realParallaxOffset = calculateParallaxAmount();

 // what is the world-space coordinate of our new point?
 intersectPointWS.xyz = vec3(vPosWS.xyz + realParallaxOffset);
 intersectPointWS.w = 1;
 // update render texture coordinates
 texCoords.xy = intersectPointWS.xz * grassTexScale;
 texCoords.z =  intersectPointWS.y * heightTexScale;
 vec2 newHFSamplePos = vHFSamplePos+convertToTS(realParallaxOffset.xz);
 // adjust height sample values
 vec2 pNearestSample = texelSize_F * floor(newHFSamplePos/texelSize_F);
 hfInterp(ivec2(round(pNearestSample*multpHF)) ,
     newHFSamplePos-pNearestSample, height_Def_F, height_UnDef_F);
 return normalMap(newHFSamplePos);
}
#endif // PARALLAX

#if defined(ADAPTIVE)
vec4 normalMap(vec2 HFSamplePos){
 vec3 fragNormalVS = getNormalVS_LinearSamp(HFSamplePos);
 return perPixelLighting(fragNormalVS);
}
// Perspective proj. based conversion from eye-space to window-space depth
// Note: The mapping is not linear, as with the perspective projection itself
float getWindowSpaceZ_Pers(float eyeSpaceZ) {
 return -0.5*(re_ProjectionMatrix[3].z/eyeSpaceZ+re_ProjectionMatrix[2].z-1.0);
}
#endif // ADAPTIVE

void main(){
 if(staticClr!=0) { FragColor = staticColor; return; }

#if !defined(ADAPTIVE)
 FragColor = perVertexLighting();
#else
 // we may modify the frag depth, set the default value.
 gl_FragDepth = gl_FragCoord.z;

 patchTexSize = textureSize(terHeightTex_Undef,0).r;
 patchTexSizeMinOne = patchTexSize-1.0;
 texelOffset_F      = 1.0/(patchTexSizeMinOne);
 multpHF            = vec2(patchTexSizeMinOne,patchTexSizeMinOne);
 texelSize_F        = vec2(texelOffset_F,texelOffset_F);

 // find the left-down corner of the cell this fragment belongs to
 nearestSample = texelSize_F*floor(vHFSamplePos/texelSize_F);
 // convert floating point to exact int sample pos
 baseTexelFetchCoord = ivec2(round(nearestSample*multpHF));

 // find the interpolation factor within the cell
```

```
    nearestSampleDist = vHFSamplePos - nearestSample;
243
    // fetch samples around the current cell
245 hfFetchCell_U(baseTexelFetchCoord,hCell_Def_U,hCell_Undef_U);

247 hCell_Undef_F  = unitY*vec4(hCell_Undef_U);
    hCell_Def_F    = unitY*vec4(hCell_Def_U);
249 height_UnDef_F = _myInterp(hCell_Undef_F,nearestSampleDist);
    height_Def_F   = _myInterp(hCell_Def_F  ,nearestSampleDist);
251
    bvec4 samplesDeformed = bvec4(
253   hCell_Def_U[0]!=hCell_Undef_U[0],
      hCell_Def_U[1]!=hCell_Undef_U[1],
255   hCell_Def_U[2]!=hCell_Undef_U[2],
      hCell_Def_U[3]!=hCell_Undef_U[3]);
257
    processingUndeformed = samplesDeformed==nonDeformedCell;
259 if(processingUndeformed){
      FragColor = perVertexLighting();
261 } else {
      #if defined(PARALLAX)
263     dx = dFdx( vHFSamplePos );
        dy = dFdy( vHFSamplePos );
265     texCoords = vTexCoords; // initialize
        FragColor = parallaxMap();
267   #elif defined(NORMALMAP)
        FragColor = normalMap(vHFSamplePos); // return;
269   #endif
      // BLENDING
271   // a blend weight of 1 for a sample means that cell has been deformed
      if(samplesDeformed!=allDeformedCell){
273     float blendFac = _myInterp(vec4(samplesDeformed),nearestSampleDist);
        FragColor = mix(perVertexLighting(),FragColor,blendFac);
275   }
    }
277 //***************************
    // ADJUST FRAGMENT DEPTH
279 if(height_Def_F<vPosWS.y && samplesDeformed!=nonDeformedCell){
      #if defined(PARALLAX)
281     vec4 posES=vec4(re_ViewMatrix*intersectPointWS);
        gl_FragDepth = getWindowSpaceZ_Pers(posES.z);
283   #else
        float eyeSpaceZ = vPosVS.z;
285     eyeSpaceZ -= abs((height_Def_F-vPosWS.y)*sin(camPitchAngle));
        gl_FragDepth = getWindowSpaceZ_Pers(eyeSpaceZ);
287   #endif

289 }
  #endif
291 // mixing the diffuse color with the terrain textures...
    FragColor *= getTextureColor();
293
  #if defined(DEFORMED)
```

```
295  if(enhanceDeform!=0) applyDeformEnhancement();
     #endif
297  }
```

Listing A.5: Heightfield Rendering Program - Fragment Shader

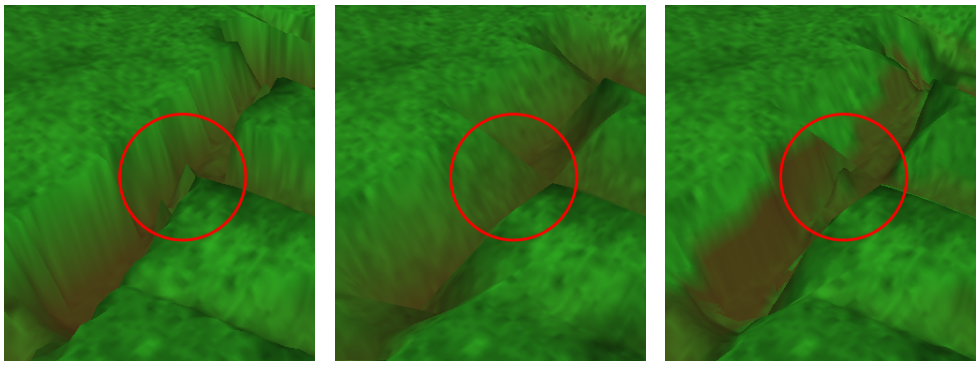
A.4 Additional Figures and Images



(a) Pass 1 (Wire-frame) (b) Pass 1&2 (Wire-frame)

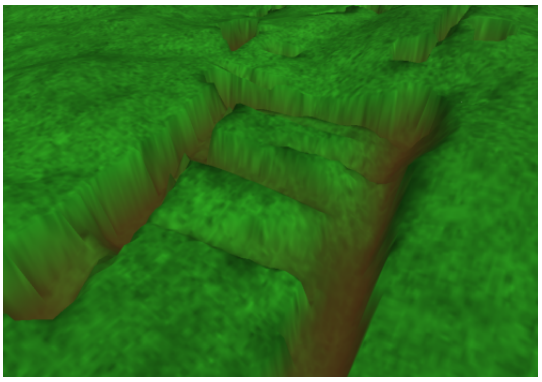
(c) Pass 1 (Filled) (d) Pass 1&2 (Filled)

Figure A.1: Sub-patch rendering applied to a deformed terrain patch

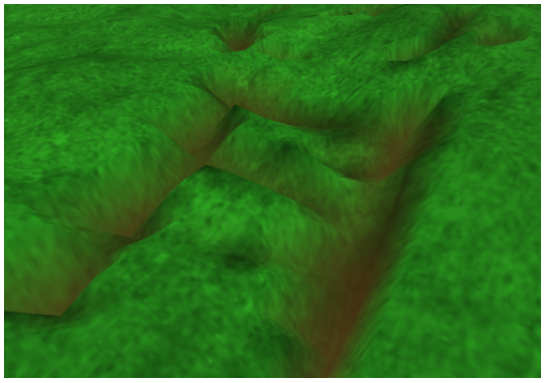


(a) Two-Step Sub-Patch (b) Single Step Simple (c) Adaptive Parallax Map.

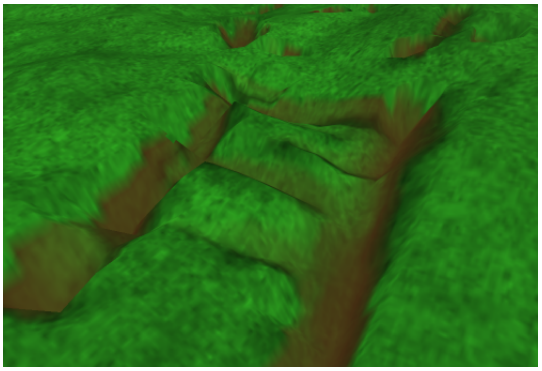
Figure A.2: Rendering of sharp features on deformed regions



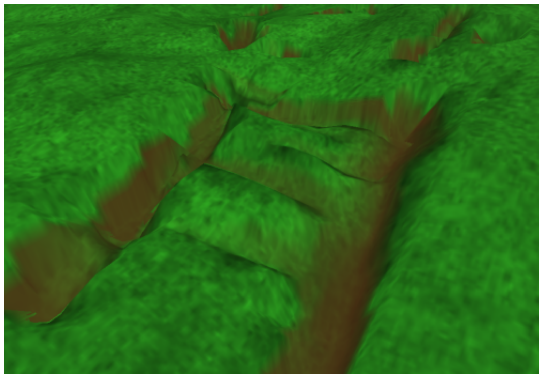
(a) Two-Step Sub-Patch



(b) Single Step Simple

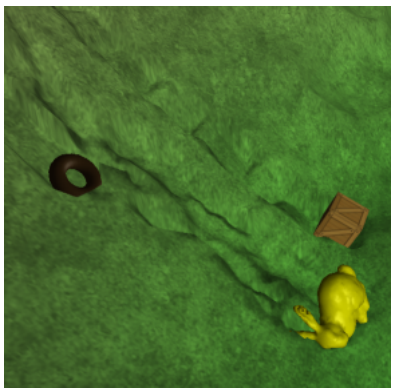


(c) Single Step Adaptive Normal-Map

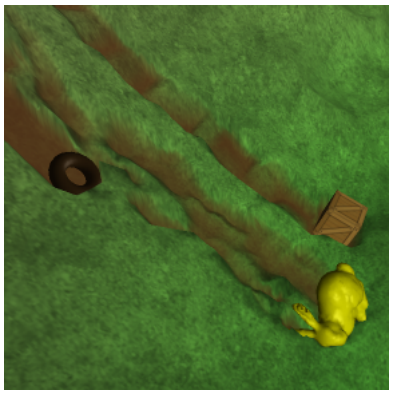


(d) Single Step Adaptive Parallax-Map

Figure A.3: The shading models for rendering deformed patches

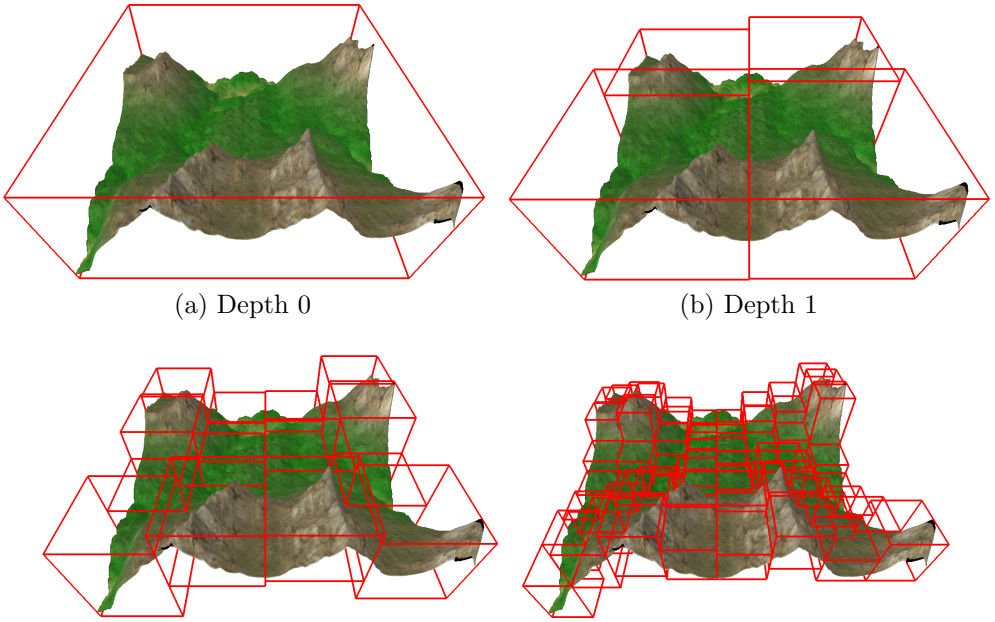


(a) No Enhancement



(b) With Color Enhancement

Figure A.4: The effect of applying deformation enhancements



(a) Depth 0 (b) Depth 1

(c) Depth 2 (d) Depth 3

Figure A.5: Quad-Tree AABB's of a sample heightfield

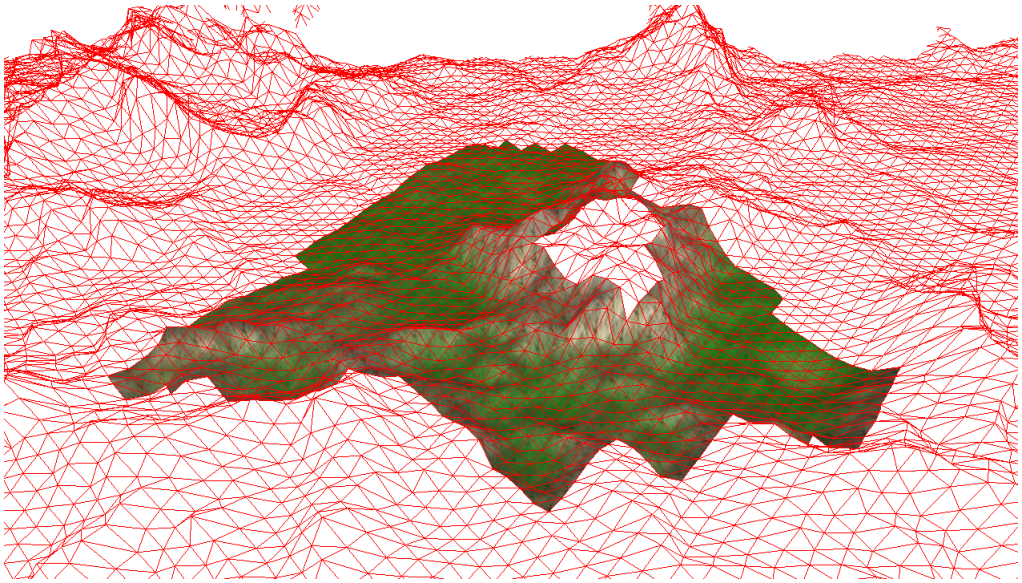
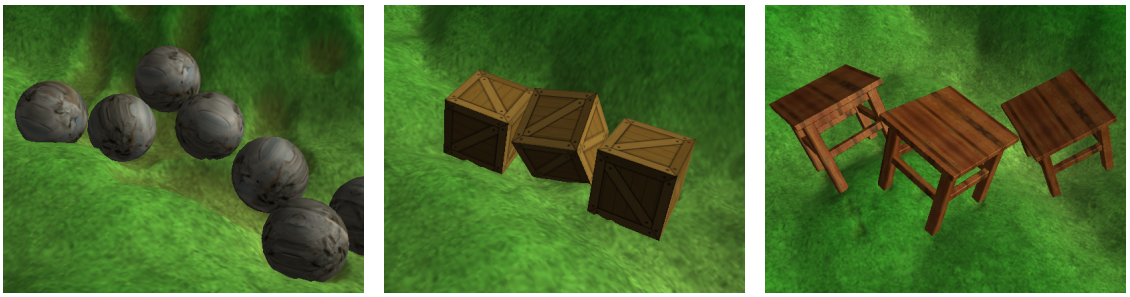
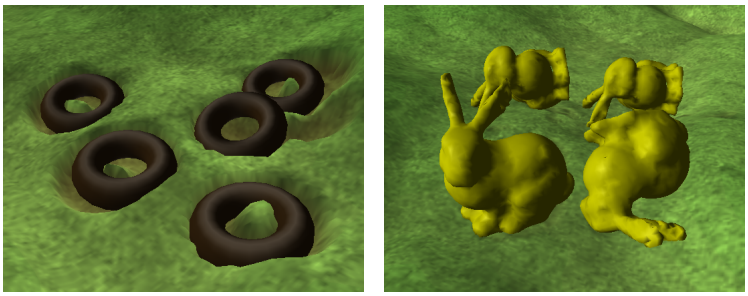


Figure A.6: 3D Frustum culling applied to terrain patches



(a) Marble (b) Crate (c) Table



(d) Torus (e) Stanford Bunny

Figure A.7: The object models that have been used to test the collision and compression pipeline

Figure A.8: Resting objects on the ground, with their collision and contact geometries

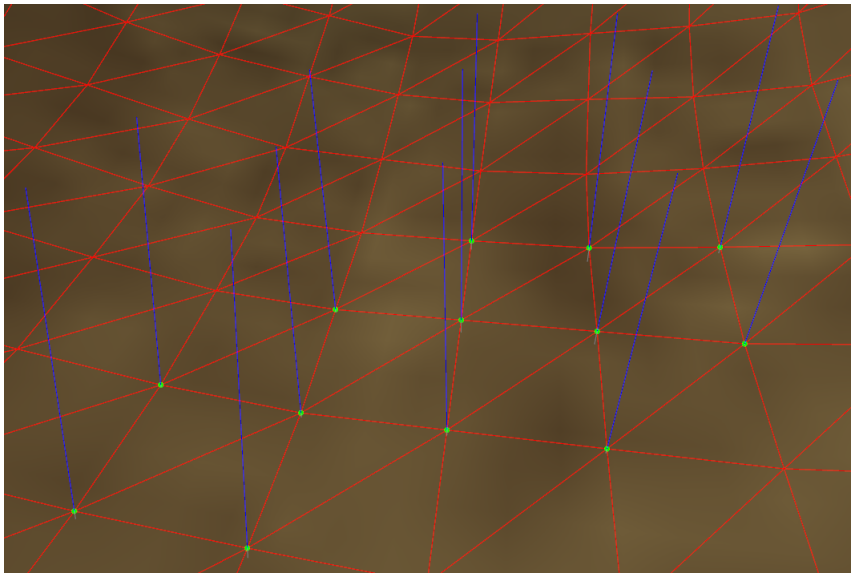
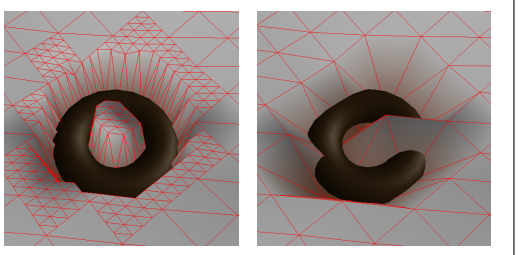
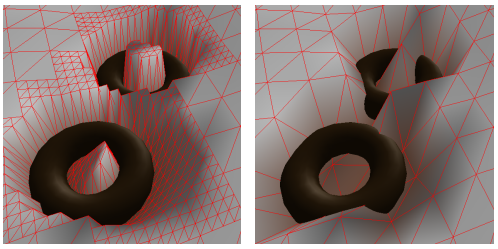


Figure A.9: The closer view of contact points and contact normals generated by a sphere-terrain intersection

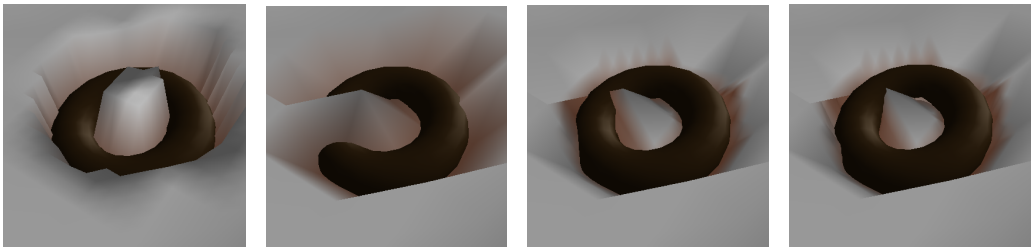


(a) Low-res tessellation covers a part of the object completely



(b) Low-res tessellation fails to capture undeformed high-resolution cells on the border

Figure A.10: Comparison of high and low resolution tessellations with interfering scene objects



(a) Sub-patch (Reference) (b) Simple (Reference) (c) Normal-Map (d) Parallax-Map

Figure A.11: The effect of adjusting fragment depth of deformed regions

Bibliography

[1] Johan Andersson. Terrain rendering in frostbite using procedural shader splatting. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 38–58, New York, NY, USA, 2007. ACM.

[2] Nguyen Hoang Anh, Alexei Sourin, and Parimal Aswani. Physically based hydraulic erosion simulation on graphics processing unit. In *GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 257–264, New York, NY, USA, 2007. ACM.

[3] Anthony S. Aquilio, Jeremy C. Brooks, Ying Zhu, and G. Scott Owen. Real-time gpu-based simulation of dynamic terrain. In *ISVC (1)*, pages 891–900, 2006.

[4] Bedrich Benes and Rafael Forsbach. Layered data representation for visual simulation of terrain erosion. In *SCCG '01: Proceedings of the 17th Spring conference on Computer graphics*, page 80, Washington, DC, USA, 2001. IEEE Computer Society.

[5] James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, 1978.

[6] Charles Bloom. Terrain texture compositing by blending in the frame-buffer. Online, November 2000. http://www.cbloom.com/3d/techdocs/splatting.txt.

[7] Benoit Chanclou, Annie Luciani, and Arash Habibi. Physical models of loose soils dynamically marked by a moving object. In *CA '96: Proceedings of the Computer Animation*, page 27, Washington, DC, USA, 1996. IEEE Computer Society.

[8] Russell Smith ODE Community. Open dynamic engine. Website, 2009. http://www.ode.org/.

[9] Robert L. Cook. Shade trees. *SIGGRAPH Comput. Graph.*, 18(3):223–231, 1984.

[10] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping. Online, October 2000. http://www.flipcode.com/archives/Fast_Terrain_Rendering_Using_Geometrical_MipMapping.shtml.

[11] Christian Dick, Jens Krüger, and Rüdiger Westermann. GPU ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, pages 43–50, 2009.

[12] William Donnelly. Per-pixel displacement mapping with distance functions. In M. Pharr, editor, *GPU Gems 2*, chapter 8, pages 123–136. Addison-Wesley, 2005.

[13] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[14] Jonathan Dummer. Cone step mapping: An iterative ray-heightfield intersection algorithm. http://www.lonesock.net/files/ConeStepMapping.pdf, 2006.

[15] Ikrima Elhassan. Fast texture downloads and readbacks using pixel buffer objects in opengl. Technical report, NVIDIA, 2005.

[16] Nico Galoppo, Miguel A. Otaduy, Paul Mecklenburg, Markus Gross, and Ming C. Lin. Fast simulation of deformable models in contact using dynamic deformation textures. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 73–82, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[17] Nico Galoppo, Miguel A. Otaduy, Paul Mecklenburg, Markus Gross, and Ming C. Lin. Dynamic deformation textures: Gpu-accelerated simulation of deformable models in contact. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 59–79, New York, NY, USA, 2007. ACM.

[18] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha. Quick-cullide: fast inter- and intra-object collision culling using graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 218, New York, NY, USA, 2005. ACM.

[19] Naga K. Govindaraju, Stephane Redon, Ming C. Lin, and Dinesh Manocha. Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 25–32, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[20] John C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12:527–545, 1994.

[21] Yefei He. *Real-time visualization of dynamic terrain for ground vehicle simulation.* PhD thesis, 2000. Supervisor-Cremer, James.

[22] Hugues 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.

[23] P. Jimnez, F. Thomas, and C. Torras. 3d collision detection: a survey. *Computers & Graphics*, 25(2):269 – 285, 2001.

[24] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. *ICAT*, pages 205–208, 2001.

[25] ONOUE Koichi and NISHITA Tomoyuki. An efficient method for displaying marks on soft grounds created by objects. *The Journal of the Institute of Image Electronics Engineers of Japan*, 32(4):328–335, 20030725.

[26] Xin Li and J. Michael Moshell. Modeling soil: realtime dynamic models for soil slippage and manipulation. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 361–368, New York, NY, USA, 1993. ACM.

[27] Ming C. Lin and Stefan Gottschalk. Collision detection between geometric models: A survey. In *In Proc. of IMA Conference on Mathematics of Surfaces*, pages 37–56, 1998.

[28] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, New York, NY, USA, 1996. ACM.

[29] Brandon Lloyd and Parris Egbert. Horizon occlusion culling for real-time rendering of hierarchical terrains. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 403–410, Washington, DC, USA, 2002. IEEE Computer Society.

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

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

[32] Morgan McGuire and Max McGuire. Steep parallax mapping. *I3D 2005 Poster*, 2005.

[33] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 41–50, New York, NY, USA, 1989. ACM.

[34] Koichi Onoue and Tomoyuki Nishita. An interactive deformation system for granular material. *Comput. Graph. Forum*, 24(1):51–60, 2005.

[35] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[36] Renato Pajarola and Enrico Gobbetti. Survey of semi-regular multiresolution models for interactive terrain rendering. *Vis. Comput.*, 23(8):583–605, 2007.

[37] Ken Perlin. An image synthesizer. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 287–296, July 1985.

[38] Fabio Policarpo and Manuel M. Oliveira. *GPU Gems 2*, chapter Relaxed Cone Stepping for Relief Mapping, pages 409–428. Addison-Wesley Professional, 2007.

[39] Fábio Policarpo, Manuel M. Oliveira, and ao L. D. Comba, Jo˙ Real-time relief mapping on arbitrary polygonal surfaces. *ACM Trans. Graph.*, 24(3):935–935, 2005.

[40] Eric A. Risser, Musawir A. Shah, and Sumanta Pattanaik. Interval mapping. Technical report, School of Engineering and Computer Science, University of Central Florida, 2005.

[41] Hanan Samet. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[42] Ari Shapiro, Petros Faloutsos, and Victor Ng-Thow-Hing. Dynamic animation and control environment. In *GI '05: Proceedings of Graphics Interface 2005*, pages

61–70, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2005. Canadian Human-Computer Communications Society.

[43] Robert Sumner, James F. O'Brien, and Jessica K. Hodgins. Animating sand, mud, and snow. *Computer Graphics Forum*, 18(1):17–26, March 1999.

[44] A. Tasora, D. Negrut, and M. Anitescu. Large-scale parallel multi-body dynamics with frictional contact on the graphical processing unit. *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics*, 222(4):315–326, 2008.

[45] Natalya Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 63–69, New York, NY, USA, 2006. ACM.

[46] Natalya Tatarchuk. Dynamic terrain rendering on gpus using real-time tesselation. In Wolfgang Engel, editor, *ShaderX7: Advanced Rendering Techniques*, pages 73–105. Charles River Media, 2009.

[47] Ondrej Štava, Bedrich Beneš, Matthew Brisbin, and Jaroslav Krivanek. Interactive terrain modeling using hydraulic erosion. In *SCA '08: ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2008.

[48] Yongning Zhu and Robert Bridson. Animating sand as a fluid. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 965–972, New York, NY, USA, 2005. ACM.

[49] Victor Brian Zordan, Anna Majkowska, Bill Chiu, and Matthew Fast. Dynamic response for motion capture animation. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 697–701, New York, NY, USA, 2005. ACM.