A GPU-Based Level of Detail System for the Real-Time Simulation and Rendering of Large-Scale Granular Terrain

by

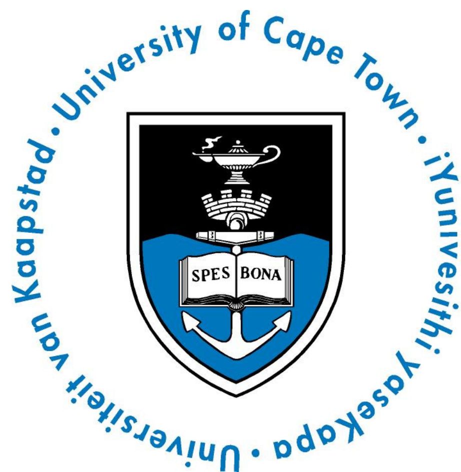
Craig Leach

supervised by

Patrick Marais

A thesis submitted for the fulfilment of the requirements of the degree of

Master of Science



May 20, 2014

Abstract

Real-time computer games and simulations often contain large virtual outdoor environments. Terrain forms an important part of these environments. This terrain may consist of various granular materials, such as sand, rubble and rocks. Previous approaches to rendering such terrains rely on simple textured geometry, with little to no support for dynamic interactions.

Recently, particle-based granular terrain simulations have emerged as an alternative method for simulating and rendering granular terrain. These systems simulate granular materials by using particles to represent the individual granules, and exhibit realistic, physically correct interactions with dynamic objects. However, they are extremely computationally expensive, and thus may only feasibly be used to simulate small areas of terrain.

In order to overcome this limitation, this thesis builds upon a previously created particle-based granular terrain simulation, by integrating it with a heightfield-based terrain system. In this way, we create a level of detail system for simulating large-scale granular terrain. The particle-based terrain system is used to represent areas of terrain around dynamic objects, whereas the heightfield-based terrain is used elsewhere. This allows large-scale granular terrain to be simulated in real-time, with physically correct dynamic interactions. This is made possible by a novel system, which allows for terrain to be converted from one representation to the other in real-time, while maintaining changes made to the particle-based system in the heightfield-based system.

We show that the system is capable of simulating and rendering multiple particle-based simulations across a large-scale terrain, whilst maintaining real-time performance. In one scenario, 10 high-fidelity simulations were run at the same time, whilst maintaining 30 frames per second. However, the number of particles used, and thus the number of particle-based simulations which may be used, is limited by the computational resources of the GPU. Additionally, the particle sizes don't allow for sand to be realistically simulated, as was our original goal. However, other granular materials may still be simulated.

Contents

List of Figures **v**

List of Algorithms **ix**

1 Introduction **1**

1.1 Research Questions 3

1.2 System Requirements 4

1.3 Contributions 4

1.4 Thesis Organisation 5

2 Literature Survey **6**

2.1 Level of Detail 6

2.2 Terrain Representations 8

2.2.1 Static Textured Geometry 8

2.2.2 Heightfields 9

2.2.3 Triangulated Irregular Network 9

2.2.4 Particles 10

2.3 Terrain LOD Techniques 11

2.3.1 ROAM 12
2.3.2 Geometry Clipmaps 14
2.3.3 Comparison 16
2.4 Dynamic Interactions 16
2.4.1 Filter-Based Methods 17
2.4.2 Momentum Based 17
2.4.3 Cellular Automata Methods 18
2.4.4 Particle-Based Granular Terrain Simulation 19
2.5 Summary 20

3 Heightfield-Based Terrain 21

3.1 Overview 22
3.2 Theory 23
3.2.1 Texture Clipmaps 23
3.2.2 Geometry Clipmaps 26
3.3 Implementation 30
3.3.1 Rendering 31
3.3.2 Updates 40
3.4 Shadows 45
3.5 Summary 50

4 Particle-Based Terrain 51

4.1 Particles and Rigid Bodies 52
4.2 Updates 53
4.2.1 Mapping of particles to 3D Grid 54
4.2.2 Update particle forces 58
4.2.3 Update rigid bodies 59
4.2.4 Update particle positions 61
4.3 Containing the Particle System 61
4.4 Rendering 62
4.4.1 Shadows 63
4.5 Scaling of Particles 65
4.6 Interactions with Models 67
4.7 Summary 68

5 Terrain Manager 69

5.1 Management of Particle-Based Terrain Systems 70
5.2 Rendering 73
5.3 Conversion from Heightfield to Particle System 73
5.4 Conversion from Particle System to Heightfield 76
5.5 Particle-Based Representation for Models 81
5.6 Summary 83

6 Evaluation and Results 85

6.1 Testing Environment 86

6.2 GPU Geometry Clipmaps 87

6.3 Particle-based Simulation 89

6.4 Integrated System 93

6.4.1 Test Scene 1 94

6.4.2 Test Scene 2 96

6.4.3 Test Scene 3 97

6.4.4 Further Tests 100

6.5 Summary 105

7 Conclusion 106

7.1 Heightfield-Based Terrain 107

7.2 Particle-Based Terrain 108

7.3 Terrain Manager 108

7.4 Future Work 109

8 References 111

A Videos 117

B Comparison of Terrains With and Without GPU Geometry Clipmaps 118

List of Figures

2.1 Example of object level of detail. 7

2.2 Heightfield with rendered terrain. 9

2.3 T-Vertex example. 12

2.4 An example of a ROAM implementation 13

2.5 ROAM diamond data structure[24]. 14

2.6 Clipmap grid example. 14

2.7 An example of a geometry clipmaps implementation. 15

2.8 Filter based terrain interactions. 17

2.9 An example of Longmore's particle-based granular terrain system. 19

3.1 An example of a terrain rendered using our geometry clipmaps implementation. 21

3.2 Example of a mipmap pyramid. 24

3.3 A comparison between a mipmap pyramid and a clipmap pyramid. 25

3.4 An example of a heightfield texture. 27

3.5 An example of a geometry clipmap-based terrain rendered with a wireframe 28

3.6 Normal Calculations for a Terrain Grid 29

3.7 Clipmap level grid structure 32

3.8 Displacement of vertices in order to form the terrain. 33

3.9 An example showing the blending regions between clipmap levels. 35

3.10 Geometry clipmaps view frustum culling. 36

3.11 Texture mapping example illustrating how textures are projected onto the terrain. 38

3.12 Clipmap position update. 41

3.13 Clipmap texture updates. 43

3.14 Coarser level sampling patterns 44

3.15 Shadow mapping camera set up. 45

3.16 Moving the near plane to cover objects behind the camera. 46

3.17 An example shadow mapping depth map. 48

3.18 Cascaded shadow maps. 49

4.1 Example of the particle-based grnular terrain simulation. 51

4.2 Particle and rigid body storage. 52

4.3 3D grid with its texture representation. 55

4.4 Collisions using a 3D grid. 59

4.5 Example particle system depth map for shadow mapping. 64

4.6 Particle system scaling. 66

5.1 Integrated system example 69

5.2 Terrain manager architecture. 70

5.3 Division of terrain for different particle scales 72

5.4 Conversion from heightfield to particle system 74

5.5 Conversion from particle system to heightfield. 77

5.6 Result of the top-down orthographic projection used to convert from a particle system to a heightfield. 79

5.7 Gaussian filter applied to the inserted terrain section. 80

5.8 Conversion of a model to a particle-based representation, using a signed distance field 83

6.1 Performance results for GPU geometry clipmaps. 87

6.2 Transition area between two clipmap levels. 88

6.3 Performance results for the particle-based simulation. 89

6.4 GPU counters captured using nVidia perfkit. 90

6.5 Performance of the individual components of the particle-based terrain simulation. 91

6.6 Longmores particle system performance results. 92

6.7 Shadowed particle system performance. 93

6.8 Screen shot of the first test scene. 94

6.9 Performance results for the first test scene. 95

6.10 Screen shot of the second test scene. 96

6.11 Performance results for the second test scene. 97

6.12 Screen shot of the third test scene. 98

6.13 Performance results for the third test scene. 99

6.14 Error present in terrain conversions. 100

6.15 Error compounded by multiple conversions. 101

6.16 Models converted to a particle representation at different scales 102

6.17 Example of the integrated system. 103

6.18 Example of the integrated system 103

6.19 Example of the integrated system used to render pebbles. 104

List of Algorithms

1 The function used to calculate the normal for each point on the terrain. Get position returns a 3D vector. 29

2 Cg shader code for calculating the blend factor for a vertex. 35

3 Cg shader code for calculating the texture coordinate for a fragment on the terrain. Note that wrapping texture sampling is used. 38

4 C++ code showing how climap levels shift. Note that this code snippet is adapted for shifting right, up and down. 40

5 C++ code snippet showing how the backplane is moved to cover any objects behind it. 47

6 C++ code snippet showing the OpenGL code used to map the particles to the 3D grid each frame. 57

7 Pseudocode to calculate texture coordinate for a particle within the 3D grid texture, corresponding to the position of the particle. 58

8 GLSL shader code snippet showing how the forces are applied to the rigid body updates. 60

9 GLSL shader code to calculate a fragments position in shadow space. . . . 65

10 Overview of the conversion from the heightfield-based representation to the particle-based system. 75

11 C++ code snippet showing the rendering order of the terrain components. 76

12 Overview of how the conversion from particle system to heightfield is processed. 78

Chapter 1

Introduction

Modern computer games and simulations feature virtual environments with hundreds of complex objects. Physics systems, such as PhysX[9], Havoc[32] and Bullet Physics[1], add realism to the scenes by allowing physically correct simulation of the interactions between these various entities. However, terrain, which represents the largest object in many of these virtual environments, has been overlooked. Terrain is often represented by static textured geometry, with no deformations taking place in response to interactions with dynamic objects. Even when deformations are supported, these interactions are low fidelity and physically implausible, relying on simple filters to modify the underlying terrain. This is particularly apparent with sandy, or granular terrain, which is extremely easy to deform in real-life, and exhibits fine grain interactions with objects.

Granular terrain displays many complex interactions, both between the constituent sand grains, and with objects. Many materials exhibit granular properties. Common examples include sand, salt, wheat and flour. Simulating granular material interactions has thus been an active area of research for many years. Particle-based granular simulations, such as that of Bell et al.[5], have been used to model complex granular interactions for industrial purposes. These simulations are extremely computationally expensive, and are often run on large computational grids. Furthermore, they lack real-time performance.

Modern consumer class graphics processing units (GPUs), have been adapted to perform generalised computations. These processors are capable of processing large amounts of data in parallel, and when utilised correctly, provide far more processing power than modern CPUs. Longmore et al.[29, 30] extended the work of Bell et al. with a GPU-based implementation, in order to create a system capable of simulating volumes of sand in real-time, on a desktop computer with a modern GPU. The resulting system is capa-

ble of simulating realistic, physically correct interactions with objects, and is capable of rendering the individual particles with realistic shading and shadowing.

Whilst this initially appears to be an attractive solution to the problem of modelling sandy terrain for games and simulations, several problems remain which limit its utility. The most important of these is computational complexity: although the system has been optimised to perform the granular simulation in real-time, it remains computationally expensive. When used with a modern nVidia GTX 770, the system is capable of simulating up to 650,000 particles in real-time. However, this represents a fairly small area of terrain. Thus, using the system to represent large-scale granular terrain, such as beaches or deserts, remains infeasible.

Heightfields are the most common form of terrain representation in modern games and simulations. Heightfields sample the terrain on a uniform grid. Each point in the grid stores the height at the corresponding point on the terrain, and represents a vertex in the corresponding terrain mesh. The terrain mesh is constructed by forming triangles between adjacent vertices. Heightfield-based terrains produce many geometric primitives. Thus, level of detail (LOD) techniques, such as geometry clipmaps[31] and ROAM[14] have been developed to reduce the number of geometric primitives required to render these terrains, thereby increasing rendering performance. Using these techniques, it is possible to render very large terrains efficiently in real-time.

We observe that heightfield-based methods are capable of rendering large areas of low fidelity terrain, whereas particle-based techniques are capable of rendering small areas of high fidelity terrain. Therefore, in order to render large-scale granular terrains in real-time, we propose a hybrid technique, which combines these two approaches.

By integrating Longmore's particle-based granular terrain system with a heightfield-based terrain system, we create a system that is capable of rendering large-scale, high-fidelity granular terrain in real-time. Interactions with the terrain take place using the particle-based simulation. The system is capable of converting between the heightfield-based and particle-based representations in real-time. Changes to the terrain in the particle-based system persist in the heightfield-based system. Our results show that the resulting system is capable of rendering large-scale granular terrain in real-time, complete with realistic physically correct interactions. Unfortunately, we are not able to achieve the appearance of sand, as the particle sizes are too large to represent sand realistically. How-

ever, other forms of granular materials can still be simulated, such as rubble or pebbles. We expect that future advances in GPU computational resources will allow finer granular materials, such as sand, to be simulated in a realistic manner.

1.1 Research Questions

There are two common methods for rendering granular terrain in real-time. However, each of them is somewhat limited.

Heightfield-based terrain is capable of representing large-scale terrains in real-time. The resulting terrain is low-fidelity, and does not exhibit realistic interactions with dynamic objects.

Particle-based granular terrain simulations are capable of simulating high-fidelity sandy terrain in real-time. These systems are computationally expensive and thus can only be used to simulate small areas of terrain.

This thesis addresses these problems by integrating a pre-existing particle-based terrain simulation with a heightfield-based terrain representation system. More specifically we address four primary research questions:

Question 1: Is it possible to combine a particle-based granular terrain simulation with a heightfield-based terrain system, whilst maintaining real-time frame rates?

We know from previous work that both of these systems may be run in real-time. However, this does not necessarily mean that if we combine them, we will maintain real-time performance. Our performance results show that this is indeed possible.

Question 2: Can we maintain changes to the terrain when converting between the two terrain representations?

We implement a top down orthographic projection to convert between the particle-based representation, and the heightfield-based representation, thereby maintaining any changes made to the particle-based system in the heightfield-based system.

Question 3: Is it possible to convert between the terrain representations without compromising the integrity of the underlying terrain?

We show in our results that the system exhibits minimal error when converting between the two different terrain representations.

Question 4: Is it possible to run multiple particle systems concurrently, of different scales, in order to simulate granular interactions at multiple points on the terrain?

Our results show that it is possible to run several simulations in real-time, each simulating a separate interaction with the terrain.

1.2 System Requirements

A set system requirements is useful in order to evaluate the resulting system, and to choose between the various potential methods available. We specify the following requirements:

- Large-scale terrains should be supported. Sandy terrains tend to be rather large (e.g. deserts and beaches), and thus large-scale support is an implicit requirement.
- The system should leverage the power of modern GPUs, in order to leave computational resources available for other parts of the system.
- Simulations performance, followed by visual performance are the most important factors, followed by simulation accuracy and finally visual accuracy.
- The system should exhibit real-time performance, so that it may be useful for games and simulations.

1.3 Contributions

Much previous work has been done in the fields of large-scale heightfield-based terrain rendering and particle-based terrain simulation. We have made the following contributions

to these fields:

- A novel texturing technique for GPU geometry clipmaps. Geometry clipmaps have traditionally been difficult to texture, and previous techniques, such as that presented by Torchelsen et al.[51], are unnecessarily complicated.
- Support for scaling of particle simulations, so that particles of arbitrary sizes may be used. Longmores system[29] only supported a single particle size. We generalise the particle size of the simulation, without the need for advanced parameter tuning.
- A method for converting between a heightfield-based terrain representation and a particle-based representation. By subdividing the terrain section, and inserting particles up to the terrain level at each point, we aim to match the height of the heightfield-based terrain with the particle-based terrain. Particles are injected over multiple frames, in order to prevent a noticeable stuttering effect.
- A method for converting between a particle-based terrain representation and a heightfield-based representation. By performing a top-down orthographic render of the particle system, and then extracting the depth buffer, the particle system can be quickly converted to a heightfield-based representation.
- Support for disabling updates of particle simulations when the contained dynamic objects come to rest. This helps to create the illusion that more particle simulations are active than is actually the case.

1.4 Thesis Organisation

The remainder of this thesis is organised as follows: Chapter 2 reviews previous work on level of detail systems, heightfield-based terrain systems and particle-based granular terrain systems. Chapter 3 presents our heightfield-based level of detail implementation. Chapter 4 examines the particle-based terrain simulation system. Chapter 5 introduces the terrain manager, which is responsible for handling the conversion between the terrain representations, and manages the dynamic objects. In Chapter 6 we present the systems results. Chapter 7 concludes the thesis, with a short summary of the findings, and suggestions for future work.

Chapter 2

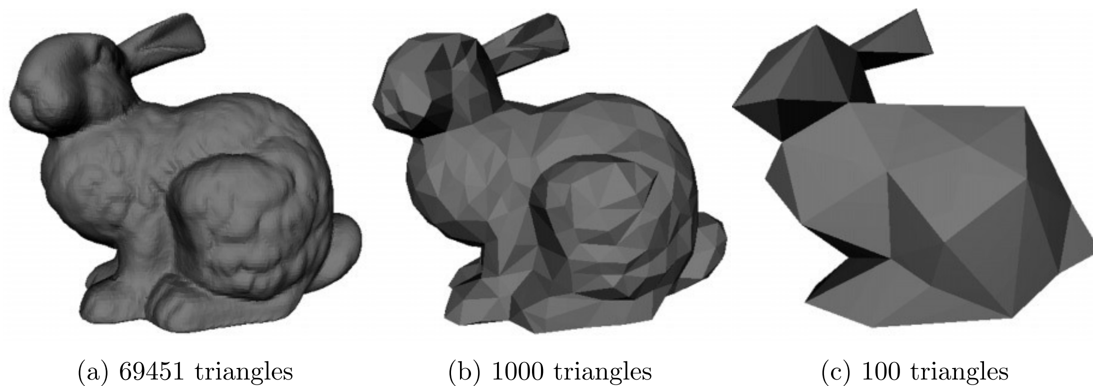
Literature Survey

Terrain forms an important part of virtual environments, and the efficient creation and rendering of terrain has thus been an active area of research for many years. Heightfield based terrains have become popular in modern games and visual effects simulations. However, these terrains display little to no dynamic interactions with dynamic objects. Recently, particle-based granular terrain simulations have emerged as an alternative to these techniques, and are aimed at simulating sandy terrain. These techniques simulate granular terrain, using particles to represent individual granules of sand. They exhibit realistic, physically correct interactions with dynamic objects. However, they have been unable to simulate large scale dynamic particle-based granular terrain in real-time, due to their computational complexity. We seek to address this by combining a particle-based terrain system, with a heightfield-based terrain system. This chapter will cover previous work on virtual terrain representation, level of detail (LOD) and dynamic interactions. It also provides a detailed analysis of competing heightfield LOD techniques.

2.1 Level of Detail

Level of detail (LOD) is the use of a different model representation of an object, with a simplified geometric structure, in order to manage rendering efficiency. Leubke et al.[33] provides an excellent survey of LOD methods. There is usually a trade off between the rendering speed, and the quality of the resulting image. Metrics are thus used to select the correct LOD for an object. This is usually based on the distance of the object relative to the viewer, but other metrics such as the object's position in screen space may also be used. Rendering efficiency is increased, as less data needs to be processed in order to

render the lower detailed representation. As an object gets further away from the observer, or further away from the users point of focus, the loss of detail becomes less evident to the user. It's important to note that the representation used for lower levels of detail needn't be the same as the main object. For instance, 2D textures (also known as impostors) are often used to represent trees in the distance.



(a) 69451 triangles (b) 1000 triangles (c) 100 triangles

Figure 2.1: The Stanford bunny rendered at 3 different levels of detail[16].

LOD schemes may be either discrete, continuous or view dependant[33]. Discrete LOD schemes rely on switching between pre-generated versions of the object. These versions may be generated by hand, or preprocessed prior to rendering and stored in memory so that they may be dynamically switched in as necessary, usually based on the distance to the observer. Continuous LOD schemes constantly refine and update a data structure, which represents the object. A geometric representation of the object, which corresponds to the required LOD, is extracted from this data structure. This generally requires less memory than a discrete LOD system, although it requires more computational resources, as the data structure which represents the object must be refined whilst the system is running. A good example of such a system is Hoppe's progressive meshes[22]. View dependant LOD schemes build upon continuous LOD schemes, by using view dependant metrics to select between various levels of detail. As the metrics are view dependant, parts of the mesh may exhibit higher values than others. View dependant LOD schemes allow for the level of detail to vary across the surface of the mesh. This is usually used when a single object may span multiple levels of detail, and is thus ideal for large objects, such as terrains.

As an object moves relative to the observer, the object must transition from one level of detail to another. This may result in a "popping" effect, as the object is suddenly replaced

with a higher or lower detailed version. Many solutions have been developed to reduce this popping effect. One example is geomorphing[23, 56], where the vertices of the model are interpolated from the one version to the other. Another option is alpha-blending between the two models. However, while these methods smooth the transitions between LOD levels, the change in representation may still be apparent to the observer.

It is also important to note that LOD systems are not limited to only reducing the number of geometric primitives used. For instance, shaders with simplified shading algorithms may be used to render the object, thereby reducing the number of operations required to render the object. Even a completely different representation of an object may be used. For instance Maciel[34] introduced "imposters", which are texture mapped quads that are used in place of the geometric representation of an object in the distance.

2.2 Terrain Representations

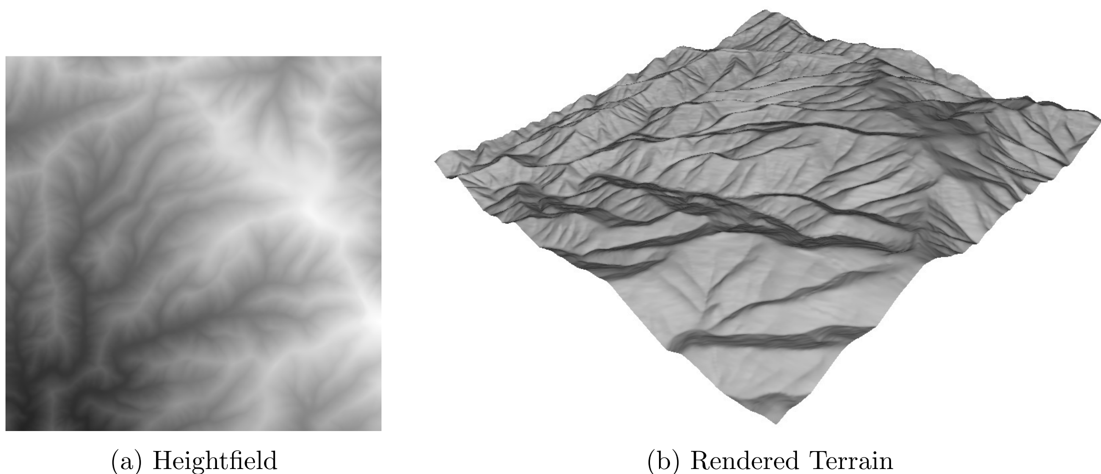
Terrain may be represented in many different ways. The choice of terrain representation affects the choice of LOD system. Additionally, some representations are better suited to dynamic terrain techniques than others.

2.2.1 Static Textured Geometry

The classical approach to terrain representation in games and simulations has been that of static textured geometry. This allows for fine detail where required, less detail for larger flat areas, and allows for complex structures, such as overhangs or caves. However, level of detail is not easily supported, as lower levels of detail need to be created by hand by an artist. The different levels of detail can then be alpha-blended together. However, this means that artists require far more time to create terrains. Additionally, dynamic terrain is not supported, as the terrain is created by an artist, and cannot be edited on the fly.

2.2.2 Heightfields

Heightfields store the height of the terrain at regularly sampled points in a grid. Thus, heightfields are often stored as greyscale images, although other formats also exist. This makes them compact and highly portable. Dynamic terrain is easily supported, as one only need alter the values stored in the grid; no other post processing is required. However, as there is only one height associated with any point on the terrain, complex structures such as overhangs cannot be represented. This may be overcome by adding static geometry at points on the terrain where these structures are required.



(a) Heightfield (b) Rendered Terrain

Figure 2.2: An example of a heightfield and its corresponding rendered terrain.

In order to render heightfields, a mesh is created that links each node to its adjacent nodes using triangles. These triangles may then be textured and lit in order to create a realistic looking terrain. However, a large amount of geometry is created, which makes them unsuitable for very large terrains. Various LOD schemes have been created in order to overcome this limitation. These will be discussed in detail in a later section.

2.2.3 Triangulated Irregular Network

Triangulated irregular networks (TINs)[40] model terrains by connecting irregularly distributed points on the surface of a terrain with triangles. More points are used in areas

of the terrain with more detail, and thus one of the major problems with grid based approaches such as heightfields, namely uniform resolution, is overcome. However, the grouping of points to create triangles is non-trivial, unlike with regular grid structures. Furthermore, when simplifying the triangle mesh of the surface for level of detail, deciding which points to remove is non-trivial, and requires preprocessing. In fact, the entire surface may need to be re-triangulated for lower levels of detail.

TINs do not adapt well to dynamic terrains. As the surface is deformed, new points are added, which affects the other triangles in the nearby vicinity. The grouping of points may need to be restructured, and preprocessing for level of detail needs to be repeated, which introduces noticeable overhead.

2.2.4 Particles

Particles may be used to represent an area of terrain. The particles collectively form a volume, in the same way grains of sand form a pile of sand. A number of particle based techniques already exist to simulate fluids. However, granular materials behave differently to fluids and require a unique set of algorithms to model their characteristics[5]. Granular materials may flow down a slope, like fluid, or form a static volume, like a solid. Traditional particle systems thus fail to faithfully recreate the complex interactions present in granular materials, and specialised systems such as those by Bell et al.[5] and Longmore et al.[29, 30] are required to model such materials.

Dynamic interactions may occur between the various particles, or with external objects. However, each individual particle need only check for collisions within its local neighbourhood, and thus exhibits $O(n)$ complexity[5]. As individual sand granules are modelled using particles, the resultant terrain has extremely high fidelity. Despite its linear complexity, a large number of particles is required to represent even a small volume of sand. Thus it is infeasible to use such a system to represent a large area of terrain.

2.3 Terrain LOD Techniques

Our technique seeks to create a level of detail system for particle-based granular terrains. This is achieved by integrating a particle-based granular terrain simulation with a heightfield-based terrain system. However, brute force approaches to rendering heightfield-based terrains are inefficient due to the amount of geometry which heightfields represent. Also, due to the high computational complexity of the particle-based terrain simulation, we seek to allocate as much computational power to this component of the system as possible. We thus seek to implement a heightfield-based LOD technique, to reduce the computational cost of rendering these terrains.

It is infeasible to render large terrains at full detail, due to the ammount of data and geometry required. Terrains typically cover large areas of the virtual world, and thus are ideal for LOD techniques, as high detail is only required in the foreground. Many LOD systems have been created to handle the different type of terrain representations. For example, ROAM[14], geometry clipmaps[31] and geomipmaps[11] have been developed for heightfield-based terrains, whereas techniques such as BDAM[7], have been developed for TIN-based terrains. Recently, there has been a great focus on adapting terrain LOD schemes to use the power of modern GPUs[45, 26, 4]. In this thesis, we will be using heightfield-based terrains, and thus our research focuses on these techniques.

With LOD schemes for terrains, different parts of the terrain exhibit higher or lower levels of detail. This may result in a vertex in a finer level of detail lying upon the edge of a geometric primitive in the coarser level of detail. This is referred to as a T-vertex(Figure 2.3). This is problematic, as it may cause differences in shading between the primitives along the edge. Additionally, due to rounding error, the vertex may not lie exactly on the edge of the coarser level primitive, the result being that the mesh does not remain "watertight" under these conditions, i.e. visible holes appear on the surface of the terrain at these intersections.

Although heightfield-based terrains are stored in a compact manner, they create large amounts of geometry. As noted above, a number of LOD systems have been created to handle this. Two of the most popular algorithms are ROAM[14] and geometry clipmaps[31]. We examine these methods in more detail, as they represent good examples of two of the most prevalent schemes for view dependant heightfield-based LOD systems: tree-based

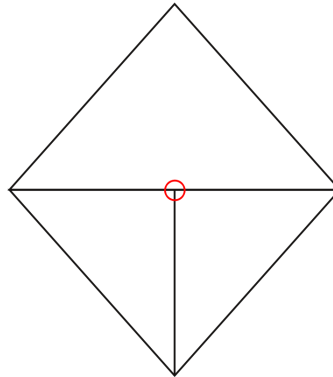


Figure 2.3: An example of a T-vertex appearing along the border of two different levels of detail.

data structures to allow for vertex decimation (ROAM) and refinement of regular grids (geometry clipmaps).

2.3.1 ROAM

ROAM[14] uses a triangular bin-tree structure to represent the terrain data, with each leaf node representing a triangle in the terrain mesh. Two priority queues are maintained; one for merge operations and one for split operations. An error is calculated for each leaf node in the tree. The heuristic used in this calculation is very flexible. For example, the distance to the camera, the surface normal relative to the camera, or the position in screen space could be used as part of the heuristic. Once the error passes a threshold, it will be added to the split queue. Those nodes with the greatest error will be split first. Adjacent nodes, which share a base edge, will first need to be split, until they are at the same level in the tree. This prevents the split operation from creating cracks along the edge which is split.

When two adjacent nodes at the same level exhibit a small enough error, they may be combined, which reduces the complexity of the terrain. ROAM allows for various error metrics to be used, such as the distance from the viewer or whether the position lies on the horizon of the terrain, and is thus very flexible.

The frame rate is directly proportional to the number of triangles that change between each frame, and thus consistent frame rates can be easily maintained. Furthermore, as only a few triangles change between each frame, popping effects are hardly noticeable, and

can be easily mitigated through the use of geomorphing. Dynamic terrain is supported, as altering the data in the bin-tree requires little in the way of preprocessing.

Triangle stripping is a technique which reduces the number of vertices required to render a set of triangles with shared vertices, by reusing previous vertices. This reduces the number of render calls required to render the set of triangles. The structure of ROAM-based terrain makes it a poor candidate for triangle stripping, as it doesn't maintain a regular grid structure. This means that many rendering calls are made, and thus the framerate is often limited by the bandwidth available on the CPU to GPU bus. Additionally, due to the triangular bin-tree structure, and the fact that the error metric varies on a per triangle basis, GPU-based implementations are infeasible.

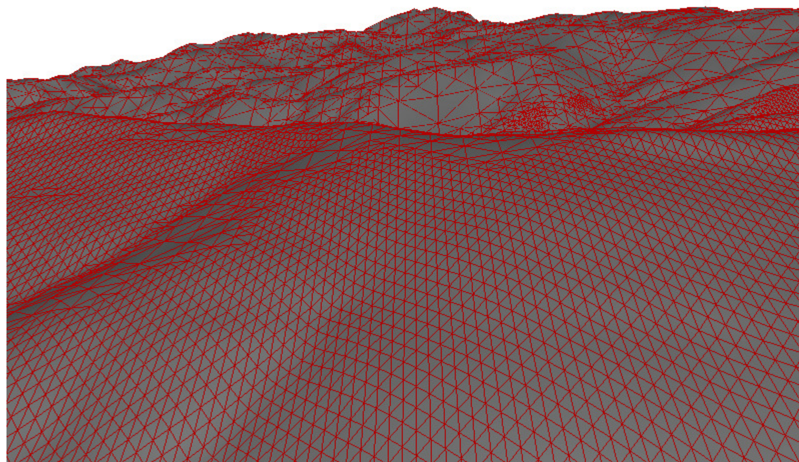


Figure 2.4: An example of a ROAM implementation

ROAM has been extended by Hwa et al.[24] to use patches of triangles, instead of refining single triangles. A single call can then be used to render each patch, which reduces the bandwidth limitation of the standard ROAM algorithm. To support this, a diamond data structure is created which centers on one node. The children of the node are then centered on the edges of the diamond, and overlap the adjacent diamond, so that when the node is refined, the mesh remains watertight. An example of this diamond data structure can be seen in Figure 2.5.

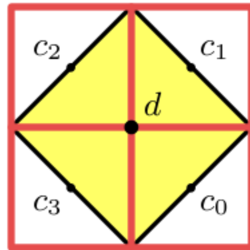


Figure 2.5: An example of the diamond data structure from Hwa et al.[24]. The center of the diamond is labelled *d*, whereas the centers of the children nodes are labelled *c0..c2*.

2.3.2 Geometry Clipmaps

Geometry clipmaps[31] uses nested regular grids to represent terrain data. The coarseness of the grids varies depending on the distance from the viewer. An example of the grid structure can be seen in Figure 2.6. As the observer moves around the terrain, the grids are refined so that the same level of coarseness is maintained at each relative distance from the viewer. A blending region is used around the outer edges of each clipmap grid. This helps to smooth the transition between the clipmap levels.

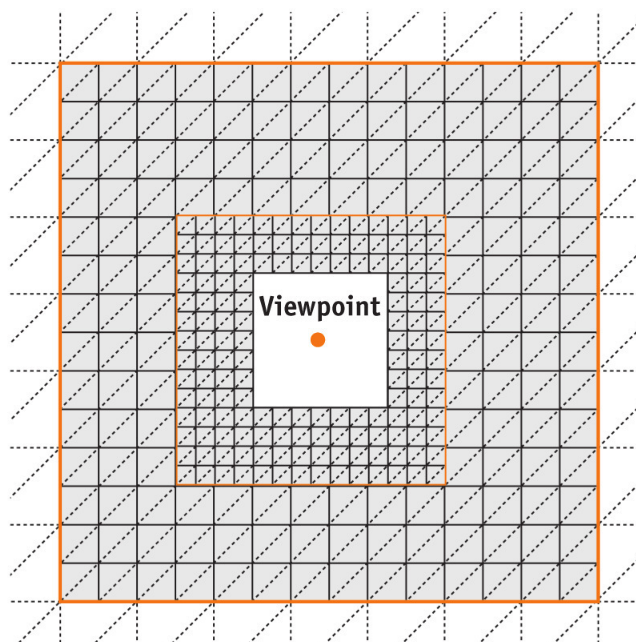


Figure 2.6: An example of 2 consecutive clipmap rings[4].

As the data is represented using regular grids, it is possible to compress the data for terrain which is further away from the observer. This results in much lower memory usage. However, it has been pointed out that if the observer moves quickly across the terrain, this may lead to a loss of image quality, as decoding the compressed data requires a considerable amount of work, and it may fall behind the rapidly moving observer.

The algorithm affords little control over the error exhibited, as the only error metric is the distance from the camera. As entire rows and columns within the grids are swapped out as the observer moves around the terrain, a visible popping effect may occur. However, this can be greatly mitigated through the use of geomorphing. As regular grids of the same size are maintained around the observer, the frame rates remain consistent. Furthermore, the grid structure allows the terrain to be very easily rendered using triangle strips, which greatly improves performance. Dynamic terrain is easily supported, as only the base heightmap needs to be deformed.

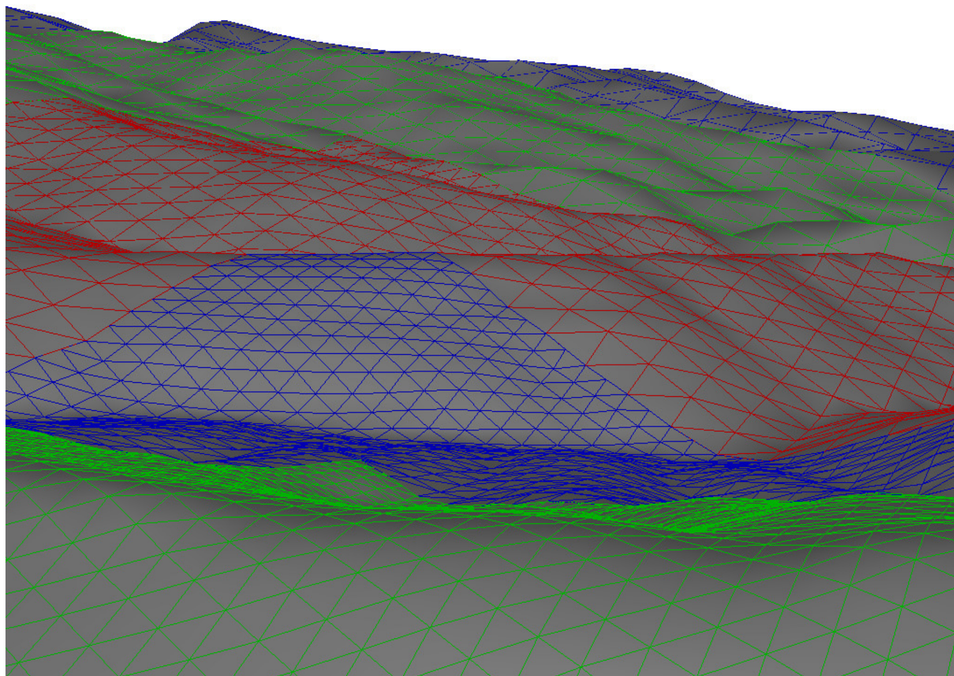


Figure 2.7: An example of a geometry clipmaps implementation.

Asirvatham and Lasasso[4] have extended the technique with a GPU-based implementation. This implementation stores a copy of the terrain heightfield on the GPU. Three

vertex buffer objects(VBOs) store grid patterns, that are then used to stitch the terrain together. Each vertex is offset to the corresponding height of the heightfield in the vertex shader. The mesh is kept watertight by blending between the values in neighbouring clipmap levels. This form of geometry clipmaps is still a view dependant LOD scheme, although it borrows concepts from discrete LODs, as the system uses the same three grid patterns to render the entire terrain.

This technique requires a minimal number of render calls, and as the heightfield and geometry is cached on the GPU, minimal bandwidth is used. Each individual clipmap is also updated by rendering updates to the clipmap textures on the GPU. This means that the technique is extremely efficient, and is generally limited by the speed of the GPU. However, updates to the heightfield still need to be passed to the GPU, in order to support dynamic terrain, and the changes then need to be applied to each of the various climap textures, which may result in framerate fluctuations.

2.3.3 Comparison

We would like our system to use GPU acceleration wherever possible, to free resources for other parts of a game or simulation. While ROAM allows for high quality terrain with flexible LOD metrics, the unfortunate lack of a viable GPU implementation means it is not well suited to our requirements. Geometry clipmaps on the other hand is easily extended to a GPU implementation, exhibits high performance, and generally consistent frame rates. We have thus selected geometry clipmaps based on these observations.

2.4 Dynamic Interactions

In order for virtual terrain to appear realistic, dynamic objects in the virtual environment should interact with the terrain in a realistic manner. Terrain should thus deform in a plausible way when objects collide with it, or move across its surface. This is especially true for granular terrain, as it deforms very easily in real life.

2.4.1 Filter-Based Methods

Dorjgotov et al.[13] created a system for real-time granular material interactions. Their system consists of a haptic device, which can be moved around in 3D space, and allows the user to interact with the virtual granular material. The granular material is deformed using a simple filter. The filter is manipulated depending on the force applied on the soil, leaving a deeper or shallower trench. A simple erosion technique is then applied to the material, which causes it to erode in a plausible way.

Figure 2.8: An example of the filter-based technique from Dorjgotov et al.[13]

While the technique is very fast and easy to implement, it completely neglects accepted models for sand deformation, redistribution and erosion, and thus the results are rather poor. Additionally, it only accounts for a ball, with no other filters present, although other filters could also be used. Finally, while it is simple to calculate the resultant force for a ball, the same is not true for complex objects.

2.4.2 Momentum Based

Zeng et al.[57] introduce a momentum based deformation system for granular terrain. Their model for the redistribution of granular material is based off work by Li and Moshell[28]. Each grid node in the heightfield maintains an associated momentum, and sand spills to

neighbouring nodes based on the direction and magnitude of the momentum. As dynamic objects collide with the terrain, force is applied to the terrain, which causes the soil to spill to neighbouring nodes. The terrain applies a reaction force upon the dynamic object, resulting in fully interactive terrain. They also introduce a rendering based collision detection model, in order to accelerate the ray casting collision detection.

The technique is relatively fast, and produces realistic results. Furthermore, it can be tuned to represent different kinds of materials, such as sand, mud and snow. However, they fail to adhere to any valid model for vertical forces acting upon the terrain, which means that the resulting deformations may appear unrealistic.

2.4.3 Cellular Automata Methods

Pla-Castells et al.[41] focus on using Cellular Automata(CA) to model dry granular systems. They seek to extend previous CA models by modelling the reaction forces and pressure distribution of the sand, thus making the terrain interactive. The terrain is modelled as a height field, and a simple set of algorithms is obtained which act only in the local neighbourhood of each cell, allowing the technique to be used in real-time simulation. The model takes into account the density of the sand, angle of repose and the deformation and redistribution of granular material resulting from dynamic collisions with the terrain. The resulting algorithms exhibit $O(n^2)$ complexity, except for the case where the pressure distribution model needs to be updated due to collisions with the terrain; this case exhibits $O(n^3)$ complexity. However, no experimental results are given, and no information is supplied about the collision detection.

Pla-Castells et al.[42] extend their previous work to use Perumpral's model[37] to accommodate objects pushing horizontally on the terrain. Experimental results are provided for the resulting system, given two different scenarios, and a variety of height field resolutions. Collision detection is implemented using the standard Open Dynamics Engine[48] collision facilities, which is based on ray casting.

2.4.4 Particle-Based Granular Terrain Simulation

Bell et al.[5] created a system to simulate granular terrain using particle based methods. The sand created using this system has a very high fidelity, and allows for realistic, physically correct interactions to occur with dynamic objects. Non-spherical particles are used in order to realistically model the granular behaviour of sand. Shear, normal and frictional forces are modelled for both collisions between the various particles in the system, and the collisions of particles with dynamic objects. While the simulation is extremely accurate, it is only feasible to simulate small volumes of sand, due to its computational complexity.

Figure 2.9: An example of Longmore's particle-based granular terrain system.

Longmore[29] extends this approach to leverage the parallel processing capabilities of modern consumer GPUs in order to simulate particle interactions, greatly improving run times. Grids are used to represent the sand particles within the system; one grid stores the positions of the particles, whilst another grid stores their momentums. These grids are used to create a 3D texture, which is passed to the GPU fragment shader, which performs the physically based simulation. However, this system may only be used for smaller-scale granular volumes, as the simulation remains computationally expensive. Additionally, due to the size of the 3D texture, memory usage is a major concern, and limits the volume of sand that can be simulated.

LOD for Particle Simulations

O'Brien et al.[39] introduced a system which allows for a simplified motion model to be used for particle simulation, which effectively creates an LOD system for particle based simulations (or "SLOD", using the papers nomenclature). Under their system, the particle based simulation is subdivided into groups of particles. Each group of particles is treated as a single granule, and the result of the interaction of this granule is applied to each of its constituent particles. However, the particle systems for which it has been implemented are rather simple, and it has not been shown that this system can be extended for use with granular terrain.

Solenthaler and Gross[49] use two discreet particle resolutions to perform fluid simulations. The coarser resolution simulates the fluid as a whole, whilst the finer resolution is only used in areas where complex interactions occur. Their system produces high quality results, while simultaneously reducing simulation complexity. However, the increase in performance is proportional to the reduction in particle count.

While both of these techniques result in much improved particle system performance, the speed up would not be sufficient to allow simulation of a complete terrain. Both techniques aim to reduce the number of particles required to simulate the particle systems, but even smaller terrains would still require far too many particles to simulate in real-time.

2.5 Summary

No LOD scheme currently exists which allows for the simulation of large-scale granular terrain. We are extending a particle-based simulation with a heightfield-based terrain system in order to create an LOD system for the real-time simulation of large-scale granular terrain. Thus we have surveyed relevant techniques in the areas of dynamic terrain simulation and heightfield-based terrain LOD systems. We have prioritised the use of GPU based techniques, in order to leverage the computational resources provided by modern GPUs. Based on our analysis of the available techniques, we have selected the GPU-based geometry clipmaps technique[4] for rendering heightfields, and Longmore's GPU-based particle based simulation[29] for fine-scaled granular simulation.

Chapter 3

Heightfield-Based Terrain

In modern games and simulations, terrains are most commonly represented using heightfields. However, due to the very large size of these terrain heightfields, level of detail systems are required in order to achieve real-time rendering performance.

Figure 3.1: An example of a terrain rendered using our geometry clipmaps implementation.

In this chapter, we present our implementation of GPU Geometry Clipmaps. The implementation is based on the work of Arvistham and Hoppe[4]. It works by rendering the terrain using concentric regular grids of increasing sizes, centred about the viewer.

The resulting system is able to efficiently render large-scale heightfield-based terrains in real-time, with minimal overhead. We analyse the inner workings of the system, and note how our system differs from the standard implementation. Furthermore, we present a unique texturing technique, which projects the textures onto the surface of the terrain, thereby simplifying the texturing of geometry clipmaps, and addressing the shortcomings of previous texturing techniques.

3.1 Overview

Geometry clipmaps represents the terrain using concentric rings, or clipmap levels, of increasing coarsness, which are centred around the observer. As the observer moves around the terrain, the clipmap levels shift, so as to maintain the same distance to the observer. In this way the terrain maintains the same level of detail, relative to the observer, as they move around the environment.

The terrain heightfield is stored as a 2D texture on the graphics card. Regular grids, which are also stored on the GPU, are used to construct the clipmap levels, and the vertices are offset in the vertex shader to match the heightmap. The heightfield for each corresponding clipmap level is updated using a rendering technique, as the observer moves around the terrain. Thus, the technique is almost exclusively GPU driven. Not only does this improve rendering throughput, as we reduce bandwidth usage on the GPU to CPU bus, but it allows us to reserve CPU resources for other computationally expensive game tasks, such as artificial intelligence and game logic.

The geometry clipmap technique performs following steps each frame:

- Update the positions of the clipmap levels.
- Update the corresponding texture for each clipmap level.
- Cull clipmap segments which fall outside the current view frustum.
- Render each clipmap level.

In the following sections, we first examine the theory behind geometry clipmaps. We

then look at the implementation of geometry clipmaps, closely analysing each individual component of the system. We point out where our system differs from the original GPU geometry clipmap technique, and introduce our new texturing technique.

3.2 Theory

Geometry clipmaps are based on texture clipmaps[50]. In order to understand how geometry clipmaps work, it is thus important to first analyse texture clipmaps.

3.2.1 Texture Clipmaps

Texture clipmaps allow large textures to be stored using a relatively small amount of memory. It draws from the technique of mipmapping[55]. Textures are represented by an array of texels. When looking at a texture displayed on a surface, the full detail of the texture is only required close to the camera. Areas of the texture that are displayed further away from the observer undergo minification; that is to say that the pixel is covered by multiple texels. The resulting colour of the pixel is thus a combination of these texel values. The texture must thus be filtered in order to produce the correct output for that pixel. As the distance from the observer further increases, more and more texels contribute to the resulting colour of the pixel, and thus we cannot use a simple bilinear filter, or other local filters. Also, due to the low sampling frequency of the texture, aliasing artefacts appear[2].

Mipmaps store prefiltered versions of the texture in order to address this problem. Each mipmap level is half the resolution of the previous level, and represents the entire texture. This creates what is commonly referred to as a “mipmap pyramid”. An example of a mipmap pyramid is shown in Figure 3.2. The sampling frequency is high enough to maintain the structure of the underlying texture. For each rendered pixel, we then choose to sample from the mipmap level which corresponds to the resolution of the output resolution. However, we cannot simply change from one mipmap level to another, as a visible seam would be easily visible by the observer. Instead, we sample the two mipmap levels which the pixel lies between, and blend the results. Filters may be used to blend between the two mipmap levels. Common examples are trilinear and anisotropic filters[2].

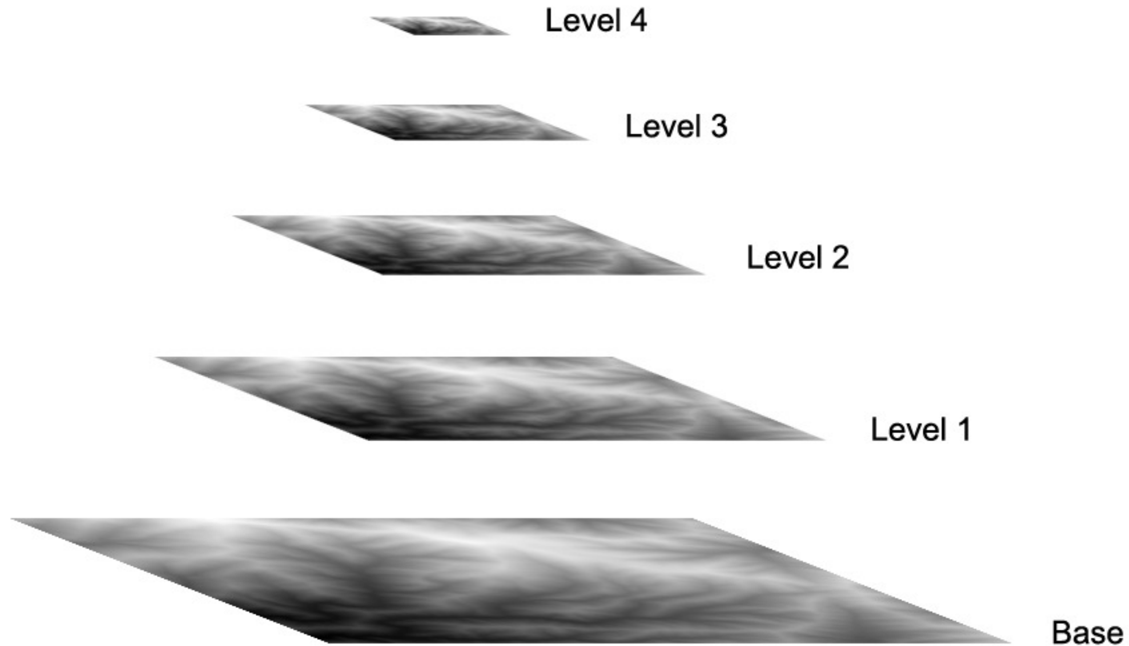


Figure 3.2: An example of a mipmap pyramid. The resolution of each successive mipmap level is half that of the previous level, and represents the entire texture.

As we are maintaining several different resolution versions of the same texture, we use additional memory. However, as each texture in the mipmap chain is half the resolution of the previous texture in the chain, the memory used to store the mipmap may be represented by the following series:

$$\sum_{n=0}^{m} \frac{1}{4^n} x$$

where m is the number of mipmap levels, and x is the memory required to store the base texture. As the sum of the infinite series is $\frac{4}{3}x$, we only require an additional 33% memory in order to store the mipmapped texture[27].

However, the memory required to store very large textures may be exorbitant, even without the additional 33% required for mipmapping. Thus, in order to store and render very large textures, an efficient alternative to mipmaps is required. As very large textures would generally cover a very large area, only a small subset of each mipmap level is sampled,

before moving on to the next texture in the mipmap chain. The memory used to store the remainder of the texture at this resolution is thus wasted, as it is never sampled.

Texture clipmaps solve this problem by limiting the size of the texture at each clipmap level. Each texture within the clipmap chain is the same size, with textures higher in the chain representing a higher resolution subset of the textures further down the chain. The surface area covered by each texture increases by a factor of 4 as we move down the clipmap chain, eventually covering the entire very large texture which we are representing. We choose a clipmap size, such that the change between clipmap levels occurs within the surface area of each clipmap level. Figure 3.3 shows an example of a texture clipmap.

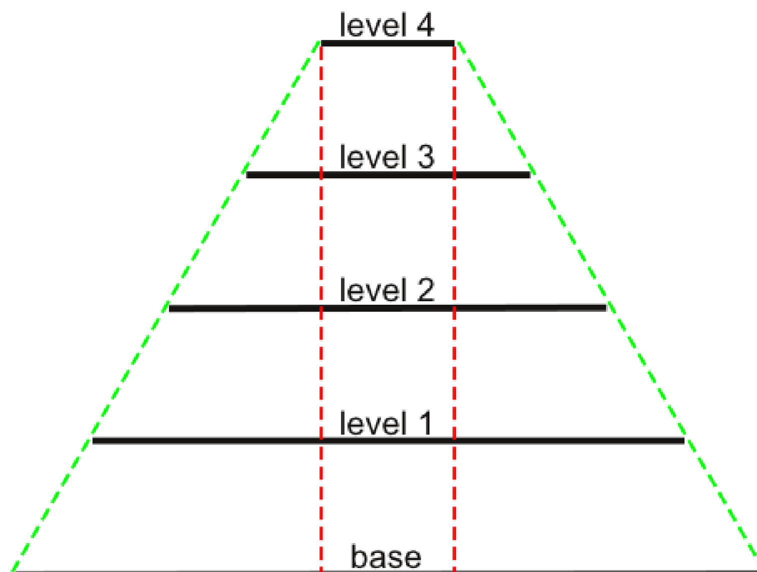


Figure 3.3: A comparison between a mipmap pyramid and a clipmap pyramid. The mipmap pyramid is outlined in green, where as the clipmap pyramid is outlined in red. As can be seen, the clipmap pyramid is a truncated version of the mipmap pyramid. Each clipmap level has the same size as the previous level, but is sampled more sparsely.

As an example, consider a texture of resolution $32,768 \times 32,768$. Assuming a 24-bit color depth, the texture requires 3.2GB of memory. However, if we represent the same texture using texture clipmaps, with the resolution of each clipmap level being 256×256, we only require 8 clipmap levels in order to represent the same texture. As the memory required to store a 256×256 texture is only 196KB, the memory used to store the entire

texture using texture clipmaps is only 1.5MB, a reduction of several orders of magnitude. Yet, the resultant rendering would appear the same, as the original image would need to be mipmapped anyway. We have thus achieved the same goal, while dramatically reducing the memory required in order to represent the texture.

The clipmap levels, and thus the texture, are centred around the observer. This is done, as the point at which the change between clipmap levels occurs is relative to the position observer. Thus, as the user moves relative to the texture, the textures in the clipmap chain need to be updated. This prevents the point at which the change in clipmap level occurs from falling outside of the area covered by the texture in the clipmap chain. However, most of the texels currently stored in the clipmap level remain within the updated texture, but translated in order to represent their position relative to the observer. In order to avoid regenerating the entire clipmap level again, each time an update occurs, we instead replace the part of the texture which is falling out of scope, with the new section which now falls within the clipmap level. We then offset the origin of the texture, which effectively means that the texels within the clipmap level shift their position, without having to move the data within memory. The texture is then toroidally addressed during sampling. Thus, when updating the textures within the clipmap level to correspond with user movement, we only need to update a small subset of the texels within the clipmap level, instead of updating the entire texture.

3.2.2 Geometry Clipmaps

Geometry clipmaps were introduced by Losasso and Hoppe[31]. Terrain is most commonly represented as a heightfield, i.e. a two dimensional regular grid of values, which represent the height at each corresponding point on the terrain. An example of a heighfield is shown in Figure 3.4. By treating the position in the array as the position on the x-z plane, and value stored in the array as y-coordinate, we thus have a list of vertices which make up the terrain. The mesh for the terrain is then constructed by creating triangles between adjacent vertices. Terrains in games and simulations are typically very large. Thus, not only does it require a significant amount of memory to store the heightfield, but it generates a significant number of geometric primitives, which is typically a limiting factor in rendering throughput.

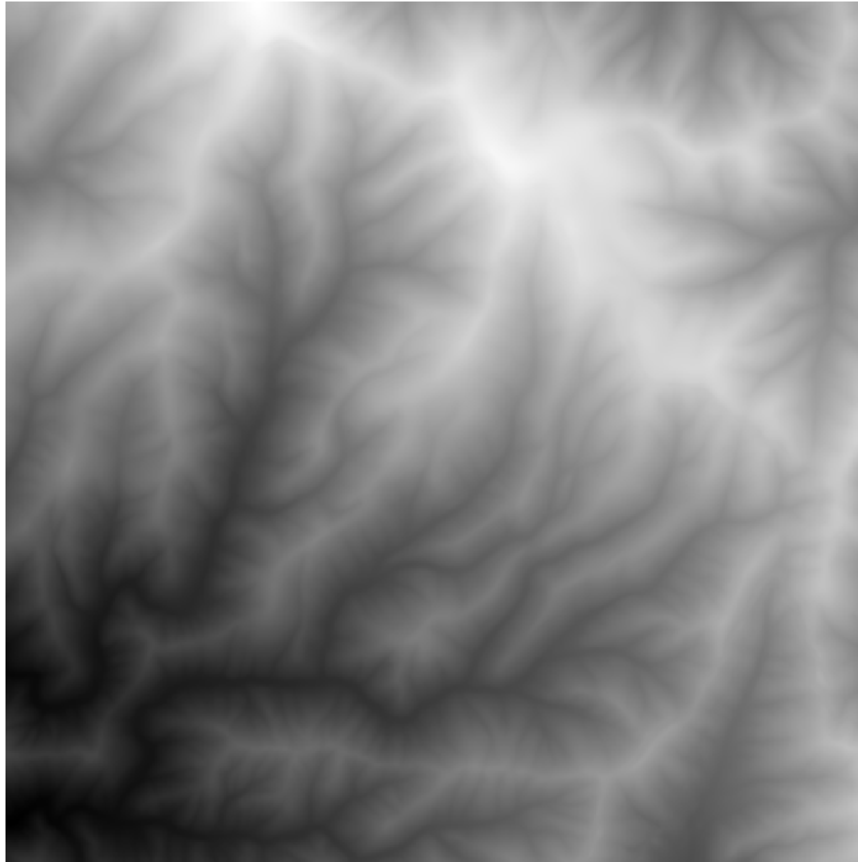


Figure 3.4: A heightfield texture. The lighter the colour, the higher the terrain height. Terrain features, such as hills and valleys, are easily distinguishable.

We can, however, extend the concept of clipmaps to heightfield-based terrains. Heightfields, thanks to their two dimensional grid representation, are essentially textures. The terrain is thus stored as a clipmap chain, with each level in the clipmap chain doubling the coarseness of the heightfield (i.e. the spacing between vertices in the x-z plane). In the same way as the density of the pixels in a texture clipmap drops as we move down the clipmap chain, so does the resolution of the terrain mesh. Figure 3.5 illustrates this concept. As the user moves around the terrain, the positions of the clipmap levels are updated, so as to maintain the same distance to the observer. Although the resultant terrain displays lower quality in areas further away from the observer, the user is generally unaware of this, as they are unable to see such fine detail so far away from them. This is because the further away from the observer a triangle is, the smaller it appears. Ideally, we would want to choose a clipmap size, such that triangles in each clipmap level maintain roughly the same size in screen space[4].

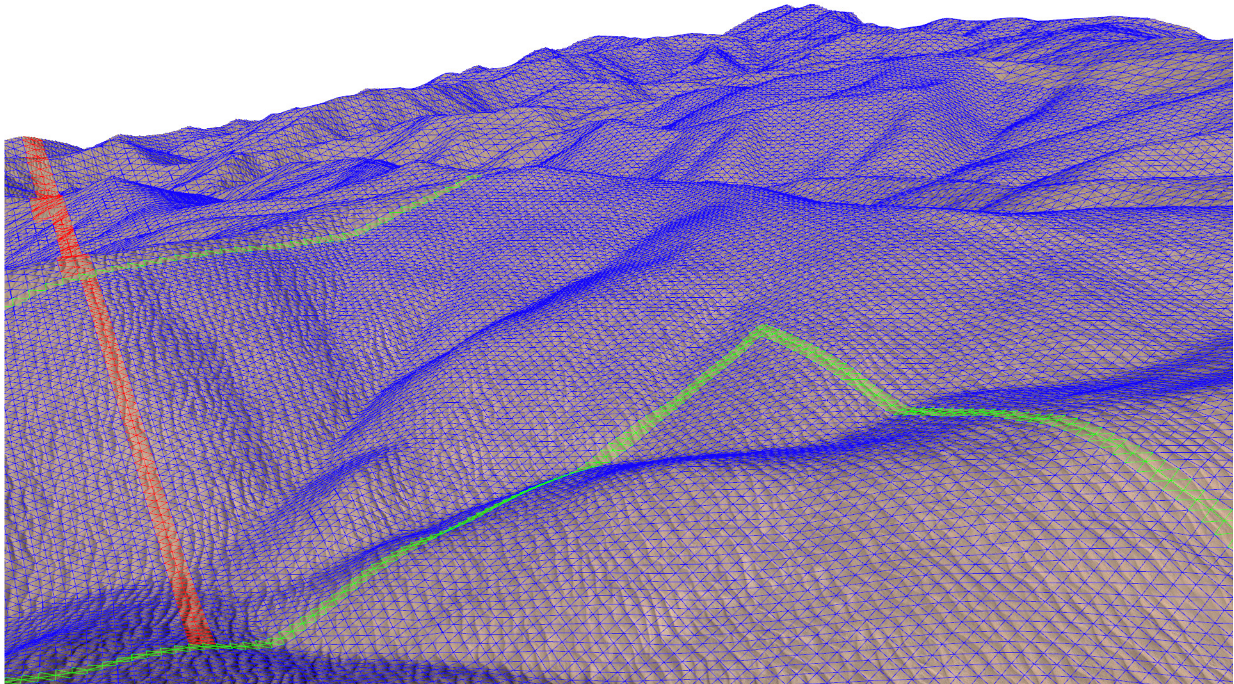


Figure 3.5: A terrain rendered using geometry clipmaps, with a wireframe, in order to show the underlying grid structure. Note that the changes between clipmap levels are easily noticeable when looking at the wireframe, but are almost impossible to spot once they are removed, due to blending, as detailed in Section 3.3.1.

As each texel within a heightmap corresponds to a vertex in the terrain mesh, by reducing the number texels within the heightmap, we are effectively reducing the number of geometric primitives required to render the terrain. Using our previous example, a terrain of size $32,768 \times 32,768$ would produce a mesh with 1,073,741,824 vertices. The number of geometric primitives is even greater: a heightmap of resolution $n \times n$ produces $2(n-1)^2 = 2,147,352,578$ triangles. Obviously, it is infeasible to render such a large number of geometric primitives, whilst still maintaining real-time frame rates.

Consider the same terrain represented with geometry clipmaps. As before, 8 clipmap levels are required to represent the entire terrain. Each clipmap level consists of $256 \times 256 = 65,536$ vertices, producing 130,050 geometric primitives. However, each clipmap level is roughly centred about the observer, and each level covers double the area of the previous level. This means that half the surface area of each clipmap level is covered by the previous clipmap level (the inner area). This is true for each clipmap level except for the first clipmap level, as there are no lower clipmap levels to cover the inner area. Thus, we require

$130,050 + 7 \times 65,025 = 585,225$ primitives, in order to represent the entire $32,768 \times 32,768$ terrain. This represents a considerable reduction in the number of polygons required. Whilst this is an extreme case, it is still possible to render the resulting terrain in real-time.

When rendering terrain, the normal for each point on the terrain must be computed to ensure correct shading of the resulting terrain. A normal map can be easily generated from the heightfield. This is done using the following algorithm (also see Figure 3.6):

Algorithm 1 The function used to calculate the normal for each point on the terrain. Get position returns a 3D vector.

function GenerateNormal(x, y)
  $tangent \leftarrow$ GetPosition($x+1, y$) $-$ GetPosition($x-1, y$)
  $tangent \leftarrow$ Normalise($tangent$)
  $bitangent \leftarrow$ GetPosition($x, y+1$) $-$ GetPosition($x, y-1$)
  $bitangent \leftarrow$ Normalise($bitangent$)
  $result \leftarrow$ Cross($tangent, bitangent$)
  return Normalise($result$)

function GetPosition(x, y)
  $result.x \leftarrow x$
  $result.z \leftarrow z$
  $result.y \leftarrow$ GetHeight(x, y)
  return $result$

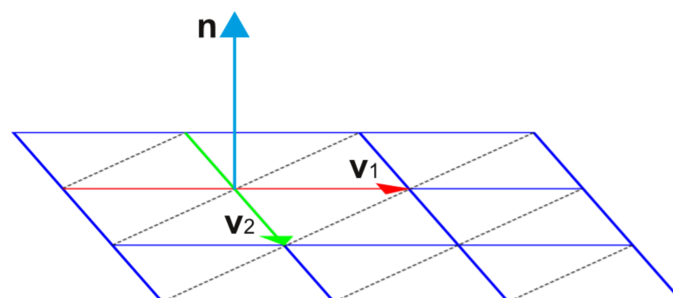


Figure 3.6: Normal calculations for a vertex on a terrain grid. $\mathbf{v_1}$ and $\mathbf{v_2}$ represent two grid aligned tangent vectors, and $\mathbf{n}$ represents the resulting normal vector.

Note that in order to prevent recalculating the normals every iteration, the resulting normal map is stored in a texture clipmap, just as with the height values.

As we move between adjacent clipmap levels, the mesh fails to remain watertight, which means that visible gaps appear between the terrains vertices. This is because the vertices of the finer clipmap level lie between those of the coarser level, and their height generally differs from the interpolated height between the coarser vertices at that point. The normals also differ, which leads to visible differences in the shading along this border. In order to avoid this, a blending region is defined along the outer edges of each clipmap level (except the coarsest level). Each texture stores both the height/normal at that point within the current clipmap level, as well as the interpolated height/normal at that point within the next coarsest clipmap level. As we approach the border of the clipmap levels, we blend between these two values, so that vertices along the outer edge exactly correspond to those in the coarser level. This eliminates any visible seams between clipmap levels.

As the terrain is rendered using concentric rings of fixed sizes, we can easily implement the algorithm on the GPU. GPU geometry clipmaps were introduced by Asirvatham and Hoppe[4]. As each additional clipmap ring is simply a scaled version of the previous ring, we need only maintain one copy of each ring on the GPU, and then scale the positions of the vertices in the vertex shader. As the user moves around the terrain, the clipmap levels shift so as to maintain the same distance from the observer, but only their positions change. Thus, we can also simply offset the positions of the vertices on the x-z plane in the vertex shader. Additionally, we can maintain the texture clipmap containing the heightfield on the GPU, and then offset the y-position of each vertex according to the corresponding height in the texture clipmap. Updates to the clipmap textures can also be processed on the GPU, leaving us with an algorithm which is almost entirely GPU-based. This is ideal, as one of the most common bottlenecks in graphics programming is the bandwidth available on the CPU-GPU bus. As the required geometry and textures are stored on the GPU, we only need a limited number of rendering calls in order to render the entire terrain.

3.3 Implementation

Section 3.2 discussed the concepts and theory behind geometry clipmaps. We now delve into the implementation specifics: We analyse the underlying components of the system

more deeply and explain the various data structures used. Our implementation is based on the work of Asivatham and Hoppe[4]. We have followed their approach, but have made slight adjustments, where appropriate, which are pointed out.

At it's core, GPU geometry clipmaps contains two major system components: the rendering component and the update component.

3.3.1 Rendering

Structure

As mentioned in the Section 3.2, the terrain is formed by a series of concentric rings. In practice, this ring is made up of smaller blocks. This is done partly to reduce memory costs, but mainly to allow for frustum culling, which we discuss later.

The grid dimension of each clipmap level needs to be an odd number. This causes the number of quads in each dimension to be an even number, which is important as it allows the grid to fit within the grid of the coarser clipmap level, while aligning with its vertices. In practice, we use a clipmap size of $n \times n$, where $n = 2^k - 1$. This has the added advantage that the grid always lies one unit off center within the coarser level grid. We explain why this is important in Section 3.3.2.

Each clipmap level is constructed from 17 smaller structures: 12 $m \times m$ grids, 4 $m \times 3$ grids, and an L-shaped grid, effectively made up of 2 $(2m+1) \times 2$ grids, where $m = (n+1)/4$. An example of a clipmap ring, with it's constituent components, is shown in Figure 3.7. The vertex data for these grids is stored on the GPU in vertex buffer objects (VBOs)[19]. VBOs cache vertex data on the GPU in an array. This means that in order to render static geometry, we don't have to transfer each vertex to the GPU every time the object is rendered, increasing overall rendering throughput. Instead, we hold a reference to the VBO, and call a render routine, which then renders geometric primitives according to the primitive type selected (e.g. triangles or quads). Index buffer objects (IBOs) are paired with VBOs, and contain a list of vertex indices. This allows a vertex to be cached once in the VBO, and used multiple times within the same render call without duplication. Alternatively, it allows for only a subset of the VBO data to be processed by a render call.

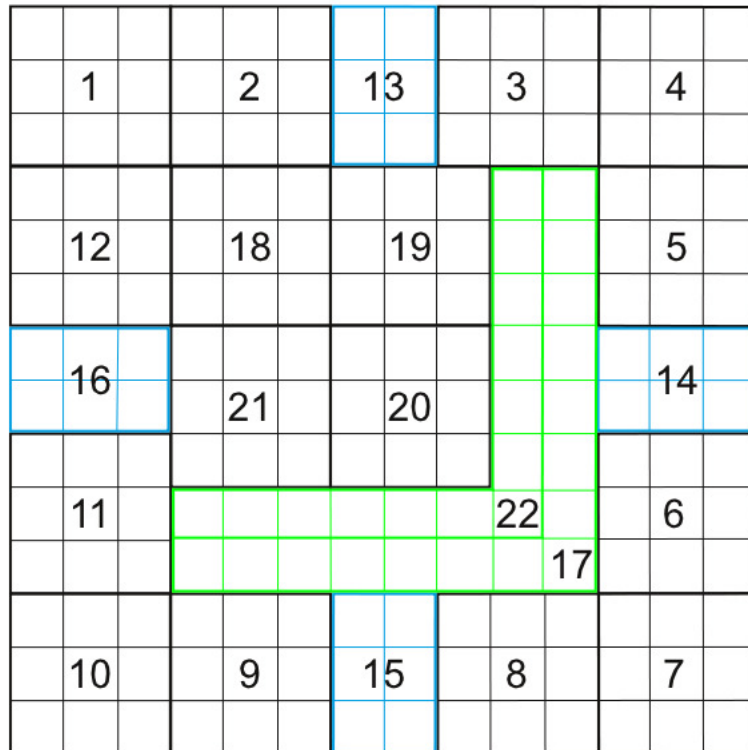


Figure 3.7: Illustration showing how a clipmap level is formed by the various grid structures. The $m \times m$ grids are shown in black, the $m \times 3$ grids in blue, and the L-shaped grids in green. Note that grids 18 – 22 are only drawn in the lowest active clipmap level.

The same set of grids is used to construct each clipmap level. We store the grids for the finest clipmap levels in the VBOs. These grids are then scaled in the vertex shader, in order to create the coarser clipmap levels. These grids create a 2D footprint of the terrain in the x-z plane. We use one VBO to store the vertices for both the $m \times m$ and $m \times 3$ grids. We use three IBOs in order to address these VBOs: one for the $m \times m$ grids, one for the $m \times 3$ grids, and one for the $3 \times m$ grids. While using slightly more memory than the original implementation, which used one IBO for all of these grids, it allows us to simplify the rendering by avoiding the rotation of the $m \times 3$ grid to create the $3 \times m$ grid. For the L-shaped region we maintain four seperate VBOs, one for each possible L-shaped grid position, and one IBO. The grids are all rendered using triangle strips, and the IBOs are set up accordingly.

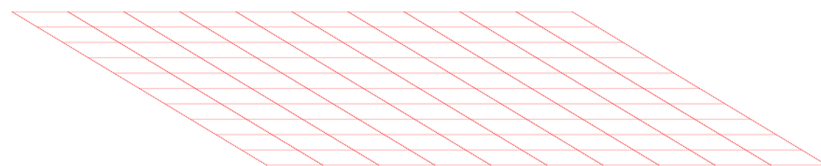
Finally, for the finest clipmap level we render the interior of the clipmap level. This

grid is made up of 4 $m \times m$ grids, and one L-shaped grid. The size of this grid is $2m \times 2m$, and thus the L-shaped grids extends beyond the bounds of this grid. To avoid this, we offset the start of the rendering by two vertices within the IBO, and terminate rendering two vertices earlier.

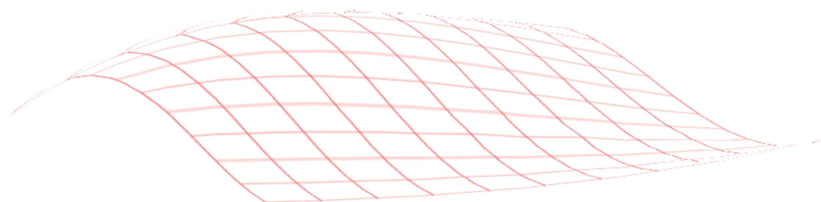
Vertex Shader

The vertex shader performs many important functions in GPU geometry clipmaps. It is responsible for scaling and translating the grids that make up the clipmap rings; displacing the vertices of the grids in the y-dimension to match the information in the heightfield clipmap textures; applying the normals required for shading to the vertices, and blending the vertices along the clipmap borders in order to keep the mesh watertight.

The position of each grid which makes up the clipmap ring is calculated on the CPU. This is then passed to the vertex shader, which applies the offset to each vertex within the grid. It also scales the grid, according to the clipmap level which the grid belongs to. These arguments are passed to the vertex shader using uniform variables. In this way, each clipmap level can be built from the same set of grids.



(a) Regular Grid



(b) Regular Grid with Displaced Vertices

Figure 3.8: The vertices of grid (a), are offset, to produce the smooth terrain surface (b).

We also pass the heightfield clipmap textures to the vertex shader. As explained earlier, the grid structure formed by the various VBOs creates a 2D footprint of the terrain in the

x-z plane, with each vertex corresponding to a texel within the heightfield texture. The texture is sampled by the vertex shader, which displaces each vertex in the y-dimension, to match the corresponding sampled value, in order to render the terrain as it is represented in the clipmap textures.

Three textures are passed to the vertex shader; one stores the height, one stores the normal and one stores the tangent. The normal and tangent are required to correctly shade the terrain (as described later in this section). The height texture is a three channel floating point texture, while the other two textures are four channel floating point textures. The normal requires two components to reconstruct, as the normal's y-component is always positive. This is because the grid representation inherent of heightfields prevents the terrain from containing overhangs. The red texture channel stores the x-component of the normal, and the green channel stores the z-component of the normal. We can then reconstruct the y-component using the following formula:

$$n_y = \sqrt{1 - (n_x^2 + n_z^2)}$$

As with the normal, the tangents x-component is always positive. Thus the tangent texture stores the y-component and z-component in the corresponding texture channels. The tangent can then be reconstructed in the same way.

In order to blend between the different clipmap levels, we require the information for the corresponding positions in the next coarser clipmap level. This information is stored in the remaining channels in each texture. A blending value is calculated, using the method shown in Algorithm 2. This algorithm results in a blending area around the outside of the clipmap grid, with `blendFactor` starting at 1.0 at the outer edge of the clipmap ring, and gradually decreasing to 0.0 over the number of vertices defined by `blendingVertexCount`. This blending factor is then used to linearly interpolate between the values stored in the texture for the current clipmap level, and those which correspond to the values for the next coarsest clipmap level. An example of the resultant blending region is shown in Figure 3.9.

Algorithm 2 Cg shader code for calculating the blend factor for a vertex.

```
//Get the number of vertices to the edge of the clipmap
float2 blend = abs( WorldPosition.xz - ClipmapCenter.xz ) / ScaleFactor;
//Reduce this by half the clipmap size. The max value is now 0
blend -= ClipmapSize / 2.0;
//Add the size of the blending area. Max size is now blendVertexCount
blend += blendVertexCount;
//Get the blending amount, between 0.0 and 1.0
blend = clamp( blend / blendVertexCount, 0.0, 1.0 );

//Blend value has been calculated in the x and z dimensions seperately.
//Take the greatest of these values as the blend factor
float blendFactor = max( blend.x, blend.y );
```

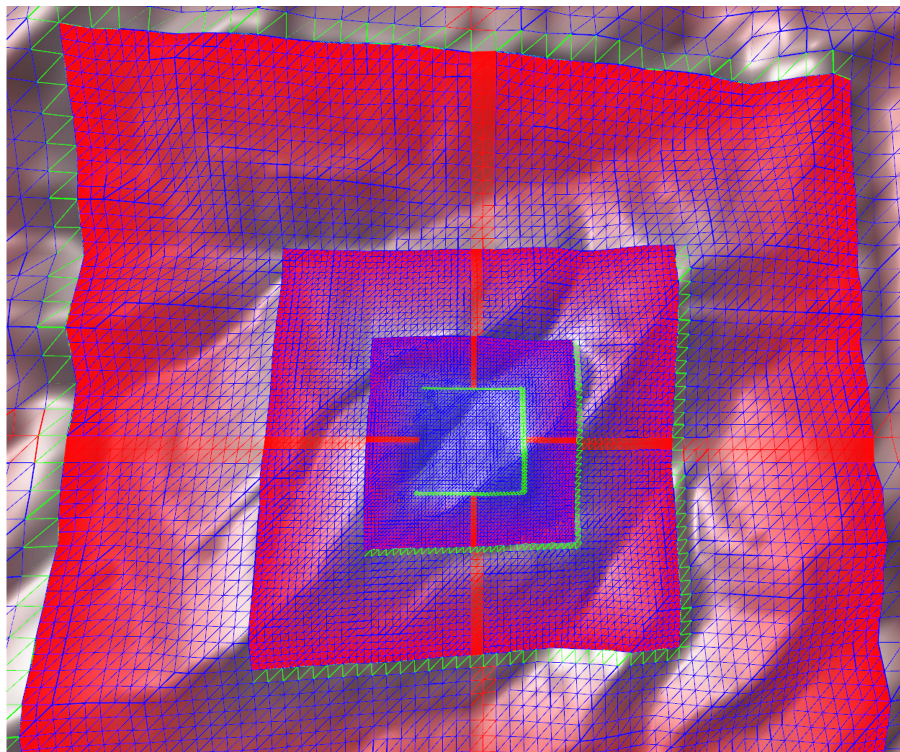


Figure 3.9: Blending region. The blending factor is represented by red colouration. As we can see from the diagram, the blending factor increases towards the edges of the clipmap level. Note that this has been exaggerated in this image, as a 63 vertex clipmap size is used, with a blending region of 16 vertices.

Frustum Cullling

As each clipmap level is centred about the observer, large portions of each clipmap ring fall outside of the view frustum. However, these sections are only discarded during the rasterisation stage of the rendering pipeline. This means that resources are wasted processing vertices in the vertex shader, and bandwidth on the CPU-GPU bus, by making rendering calls which do not contribute to the final rendered image. It is therefore far more efficient to cull these regions, so that the rendering calls don't take place if that section of terrain falls outside of the frustum.

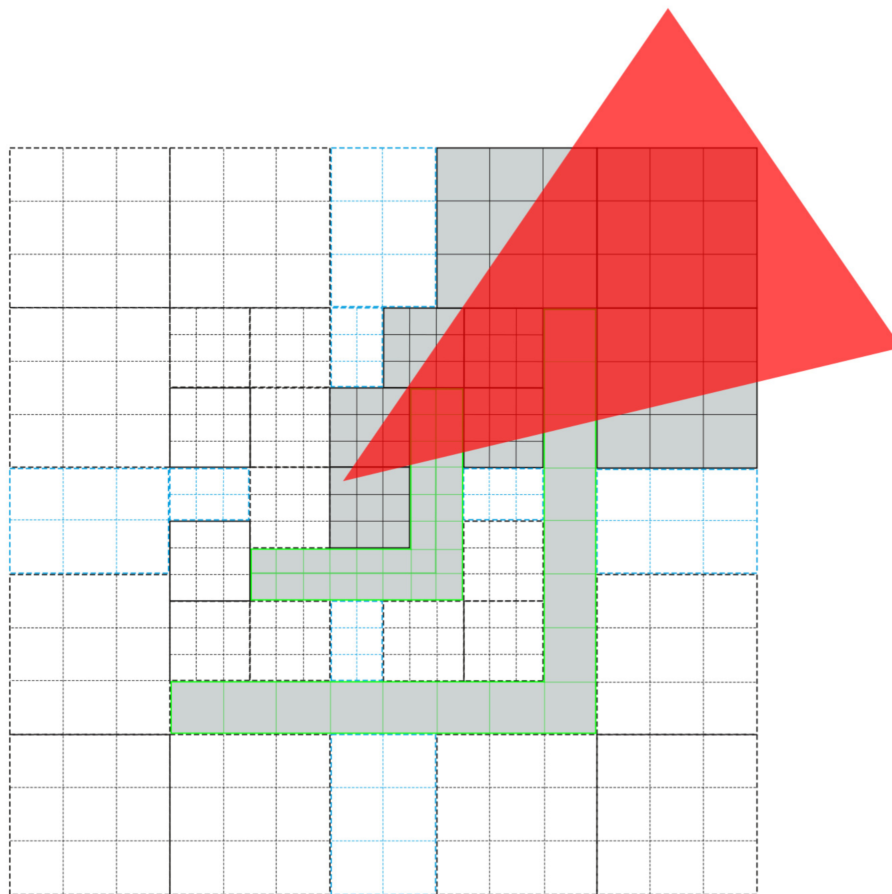


Figure 3.10: View frustum culling for geometry clipmaps. The view frustum is outlined in red. Rendered grids are shown in grey, whilst culled grids are dashed.

Now that each clipmap level is constructed from smaller grids, we can cull these grids against the view frustum. We know the bounds of each grid in the x-z plane, but unfortu-

nately we would have to sample the clipmap textures in order to ascertain the grids bounds in the y-dimension. This is undesirable, as it would consume bandwidth on the CPU-GPU bus. So instead, the maximum and minimum heights in the terrain are stored, and these are used in conjunction with the bounds in the x-z plane in order to create a bounding volume, against which the frustum culling is evaluated, as in the original GPU geometry clipmaps implementation[4].

In order to cull the grids, we first get the matrix for the view frustum. From this, the view frustum planes are extracted. We take the position on each corner of the bounding volume of the grid, and evaluate these positions in relation to the planes. If all the sampled points fall outside of the same plane, this means that the grid falls outside of the view frustum, and thus may be culled. Unlike the original GPU geometry clipmap implementation, we test each grid against the view frustum, including the $m \times 3$ and L-shaped grids (the original system only tested the $m \times m$ grids).

Texture Mapping

The original GPU geometry clipmaps implementation did not provide a method for texturing the resulting terrain. Geometry clipmaps are indeed difficult to texture, as a texture may be tiled multiple times between vertices at coarser clipmap levels. Torchelsen et al.[51] introduced a texturing technique which assigns a texture coordinate to each vertex within the grid. A value is obtained, which defines how many times the texture is repeated between vertices in the grid. This then allows the correct texture coordinate to be inferred for each fragment, allowing texturing of the GPU geometry clipmap based terrain.

We found this approach to be overly complicated however, as it added yet another set of vertex attributes which need to be maintained. Instead, we observe that if a texture is tiled across the surface of the terrain, the texture coordinate for each fragment is related to its position on the x-z plane. This is possible, as the surface of the sandy terrains which we are simulating would generally look similar everywhere, and thus a single texture can be tiled across the surface. The following line of code thus produces the correct texture coordinate for each fragment:

Algorithm 3 Cg shader code for calculating the texture coordinate for a fragment on the terrain. Note that wrapping texture sampling is used.

```
//texRepetition = the number of units in world space over which the texture
//should repeat.
float2 texCoord = worldPosIN.xz / texRepetition;
```

Using this technique, we are effectively projecting the texture onto the terrain. As it simply relies on the position of the fragment in world space, the resolution of the grid has no impact on the resultant texture coordinate. Note that we allow the texture coordinates to wrap around in the texture sampler. This is much simpler than the previous method, as we do not need to worry about maintaining an additional set of vertex attributes. This technique is illustrated in Figure 3.11.

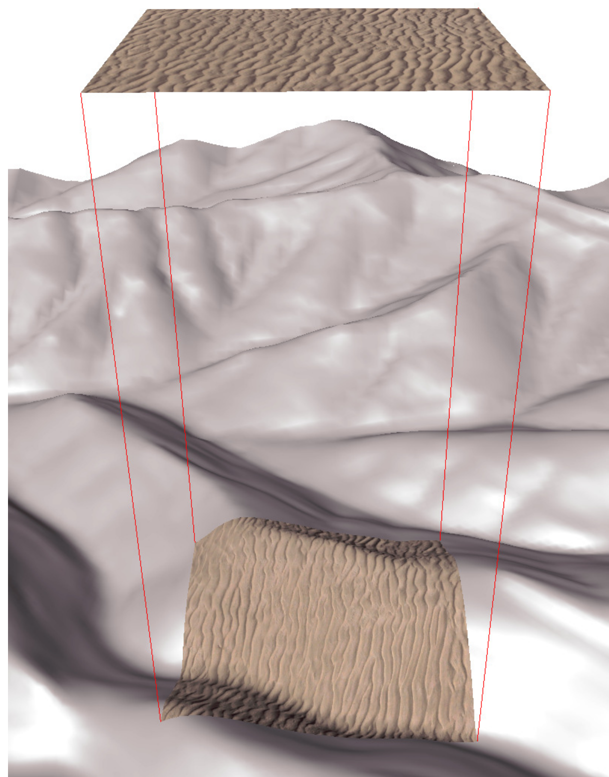


Figure 3.11: Texture mapping example illustrating how textures are projected onto the terrain. The texture coordinates are based on the position of the fragment in the x-z plane. The texture is repeated across the surface of the terrain, resulting in a smooth, textured appearance.

Bump mapping is a technique which encodes detailed information about the surface of a geometric primitive in a texture. The idea was originally introduced by Blinn[6]. In particlular, we employ normal mapping, using the technique of Cook[8], which has become the de facto standard for such techniques. This technique allows the normals to vary across the surface of a geometric primitive, thus mimicking the appearance of more detailed geometry.

In order to achieve this, the inverse tangent space matrix is calculated. This allows for a normal to be transformed from object space into world space. In order to calculate the inverse tangent space matrix, we require the normal, tangent and bitangent for the vertex. These are the same normal, tangent and bitangents that we have already calculated, as in Figure 3.6. The normal and tangent are passed into the vertex shader, as discussed earlier in this section. The bitangent is orthogonal to these two vectors, and may thus be calculated as $\mathbf{b} = \mathbf{n} \times \mathbf{t}$. As shown by Lengyel[27], the inverse tangent space matrix may then be constructed as follows:

$$\begin{bmatrix} \mathbf{t}_x & \mathbf{b}_x & \mathbf{n}_x \\ \mathbf{t}_y & \mathbf{b}_y & \mathbf{n}_y \\ \mathbf{t}_z & \mathbf{b}_z & \mathbf{n}_z \end{bmatrix}$$

This is performed on a per fragment basis, with the normal, tangent and bitangent being passed from the vertex shader to the fragment shader. They are thus interpolated across the surface of the triangle, leading to a smoothly varying inverse tangent space matrix.

Once the inverse tangent space matrix is calculated, the normal for the fragment is sampled from the normal map. The normal map is a texture, which represents the normal of the surface at the corresponding point. This normal is in tangent space, and thus by transforming this normal by the inverse tangent space matrix, the normal is transformed into world space. Lighting calculations are then performed using this transformed normal, resulting in a realistic looking surface, with normals varying across the surface. This is an efficient technique, requiring only a single texture lookup, and produces believable looking surfaces.

However, this technique does have certain limitations. The texture may appear stretched in areas with a steep gradient, as primitives may appear larger in texture space than in world space. Also, it doesn't allow for artists to have finer control over the texture coordinates. However, for our scenario (i.e. sandy terrain), neither of these issues are problematic.

3.3.2 Updates

Geometry Updates

As the user moves around the terrain, the position of the clipmap levels need to shift, in order to maintain the same relative distance to the observer. This is computed on the CPU. The resulting positions of each grid within each clipmap level is then passed to the vertex shader. This is where our choice of $n = 2^k - 1$ as the clipmap size becomes relevant. As noted earlier, this causes each clipmap level to be offset by one unit, both horizontally and vertically, within the coarser level. The L-shaped grid is used to fill this area.

Algorithm 4 C++ code showing how climap levels shift. Note that this code snippet is adapted for shifting right, up and down.

```
void ClipmapShift( const int level, const Direction direction )
{
    //Move Left
    if( direction == LEFT )
    {
        m_clipmaps[level].left = !m_clipmaps[level].left;

        //if the clipmap position has changed to the left, that means the grid
        //has shifted within the coarser level. Grid shifts by 2 units.
        if( m_clipmaps[level].left )
        {
            m_clipmaps[level].newClipmapPositionX -= 2.0f *
                m_clipmaps[level].scale;
            m_clipmaps[level].newTextureOrigin.u -= 2.0f / (float) (
                c_clipmapSize + 1 );
            if( level + 1 < c_clipmapLevels )
            {
                //Gid is has either shifted 1 unit to the other side, or is
                //now 2 units offcenter. Either way, the coarser level must
                //be updated to reflect this.
                ClipmapShift( level + 1, LEFT );
            }
        }
    }
}
```

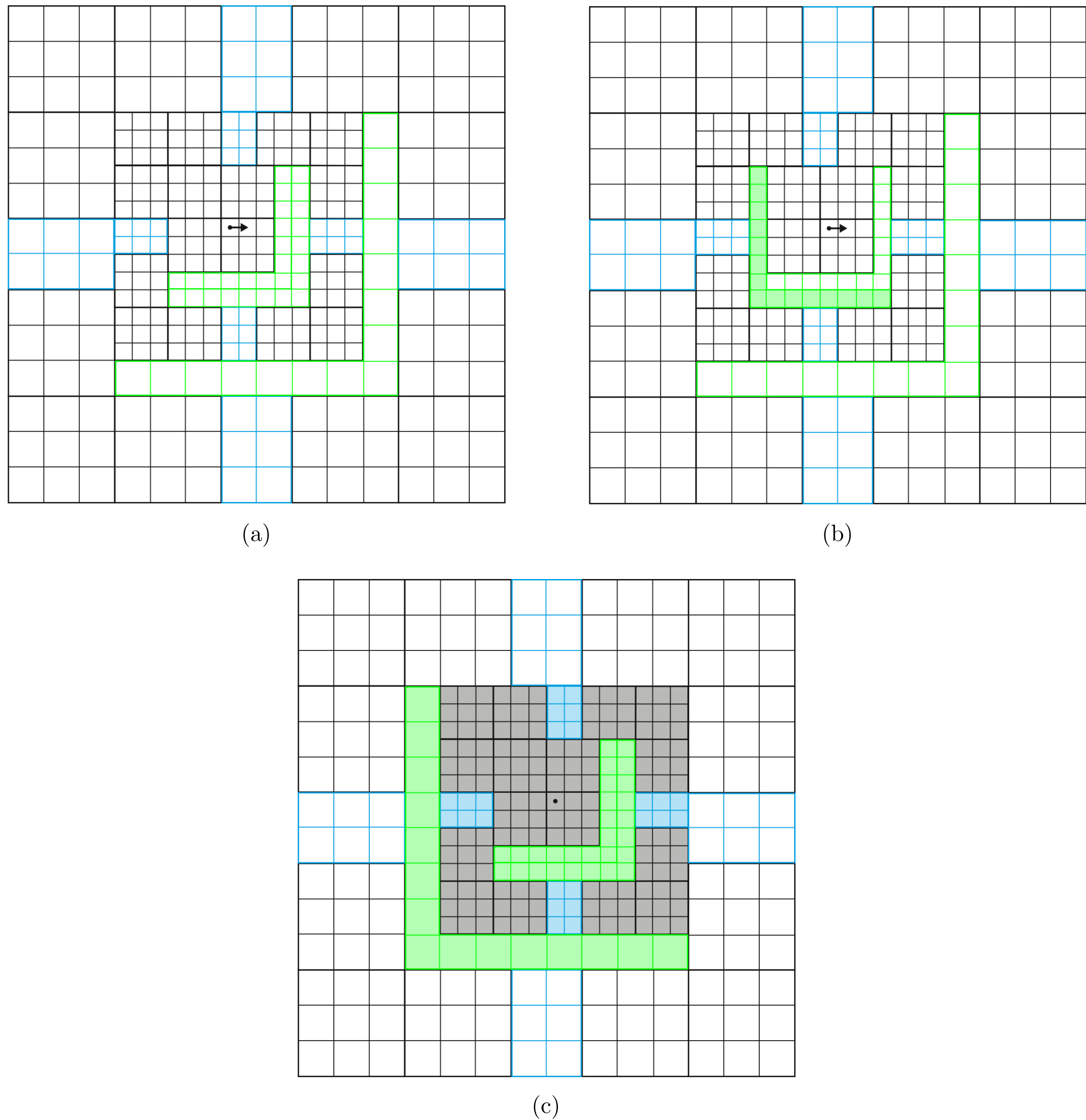


Figure 3.12: Illustrates the updates to the clipmap structure, as the observer moves across the surface of the terrain. In the diagram, the observer moves 2 units to the right. The grids which shift are highlighted. In (b), the L-shape switches to the opposite side to adjust for the new offset within this level. In (c), the observer is now offset 2 units within the lower clipmap level. Thus, the clipmap level shifts to the right, into the space occupied by the L-shape in the coarser level. This L-shape then switches to the other side, as the observer is now offset by 1 unit to the right within this clipmap level.

Clipmap levels always shift two units, since they have to move two units in order to move one unit within the coarser level clipmap level. As the user moves, the finer clipmap level eventually becomes offset by two units within the coarser level. When this occurs, the coarser level shifts two units (i.e. one unit in the next coarser level) in that direction, and the L-shaped grid moves to the other side. Note that if the user moves towards the L-shaped region, the finer clipmap level becomes offset by one unit in the other direction, and the L-shaped grid then moves to the other side of finer clipmap level.

This means that a clipmap level only needs to be updated once the finer clipmap level has moved two units within it's area. The coarser level clipmaps are thus rarely updated. Consider for example a clipmap chain with 7 clipmap levels. Assuming that the vertices at the finest level are placed 1 meter apart in the virtual world, the coarsest level would only shift once for every 128 meters of user movement in a dimension. This is because the unit size of the coarsest level is 64 meters × 64 meters, and the finest level needs to move two units within the coarsest level in order for the coarsest level to shift.

Texture Updates

As the clipmap grids shift, the clipmap textures, which correspond to the vertices in the grids, need to be updated. As in the original GPU geometry clipmap system[4], we employ a rendering based update system in order to maintain the clipmap textures. This is ideal, as it requires very little CPU input.

As most of the data in the clipmap texture remains in the texture, we simply update the section which has fallen out of scope, with the data from the new area which we are moving into. This prevents us from having to translate the positions of the existing data within the texture clipmap. The textures are toroidially addressed (i.e. we wraparound the texture borders when sampling the textures), so we simply adjust an offset, which the vertex shader uses when sampling the texture.

The update to the texture is achieved by rendering texture mapped quads directly to the clipmap texture. This is done for movement in both the x and the z dimensions. The quad begins at the edge of the clipmap texture which corresponds to the direction of the clipmap shift, and extends so as to cover the area by which the clipmap shifts. Note, that this may cause the quad to extend beyond the bounds of the texture. In that case, a

second quad is used, beginning on the other side of the texture, and extending to cover the remaining area, thereby effectively recreating the wrap around, which is used to sample the texture.

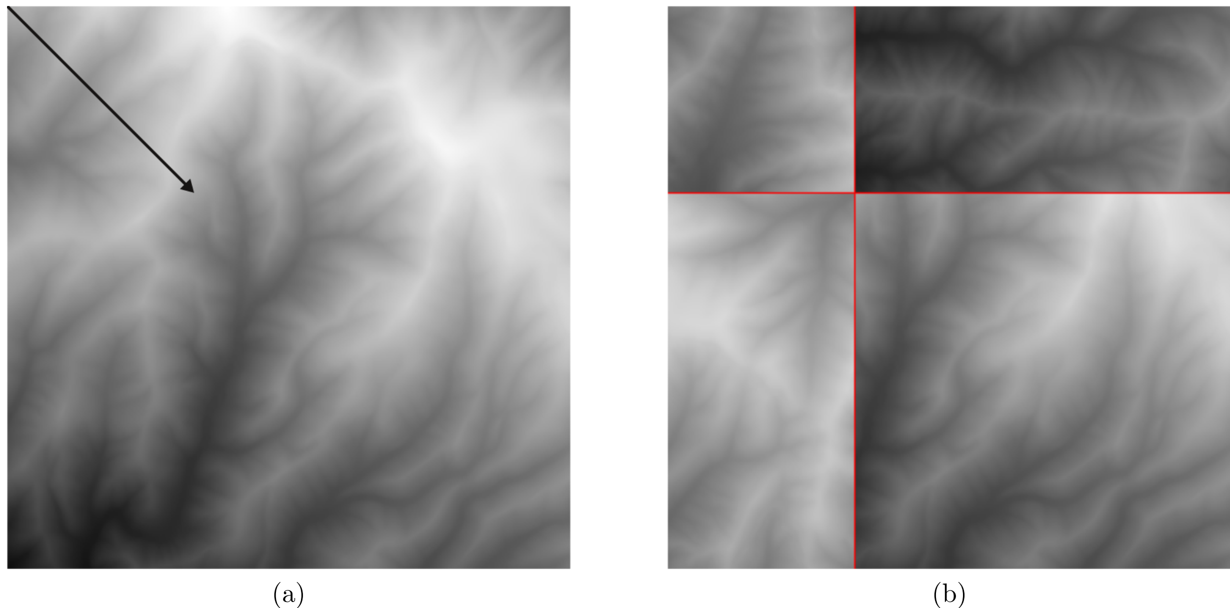


Figure 3.13: The clipmap texture is updated to correspond with clipmap movement. The arrow in (a) shows the movement of the observer, and (b) shows the updated texture. The new texture origin is denoted by the crossing point of the two red lines. The texture is toroidally addressed, resulting in a smooth texture, with no discontinuities.

Multiple render targets are used, to update both the texture clipmap for the current level, as well as the texture clipmap which corresponds to the data for the next coarsest level. The quads are textured with data from the underlying heightfield, normal maps and tangent maps for the current clipmap texture, whereas for the texture which corresponds to values from the coarser clipmap level, we sample the clipmap texture of that coarser level.

When the coarser level is sampled, we sample two texels, and interpolate the result. This is a simpler method than presented in the original implementation, but the results are equally valid. The texels sampled depends on the position of the pixel within the clipmap, as those pixels in odd positions correspond to vertices which lie between vertices in the coarser clipmap level. Pixels at even positions align with the vertices at the coarser level, and so only one sample is needed. Also remember though, that we need to consider their position in each dimension. So, if a pixel is odd positioned in either dimension, two

samples used. If it is even in both dimensions, the same vertex is sampled twice. Figure 3.14 illustrates how the coarser level is sampled for each possible even/odd configuration.

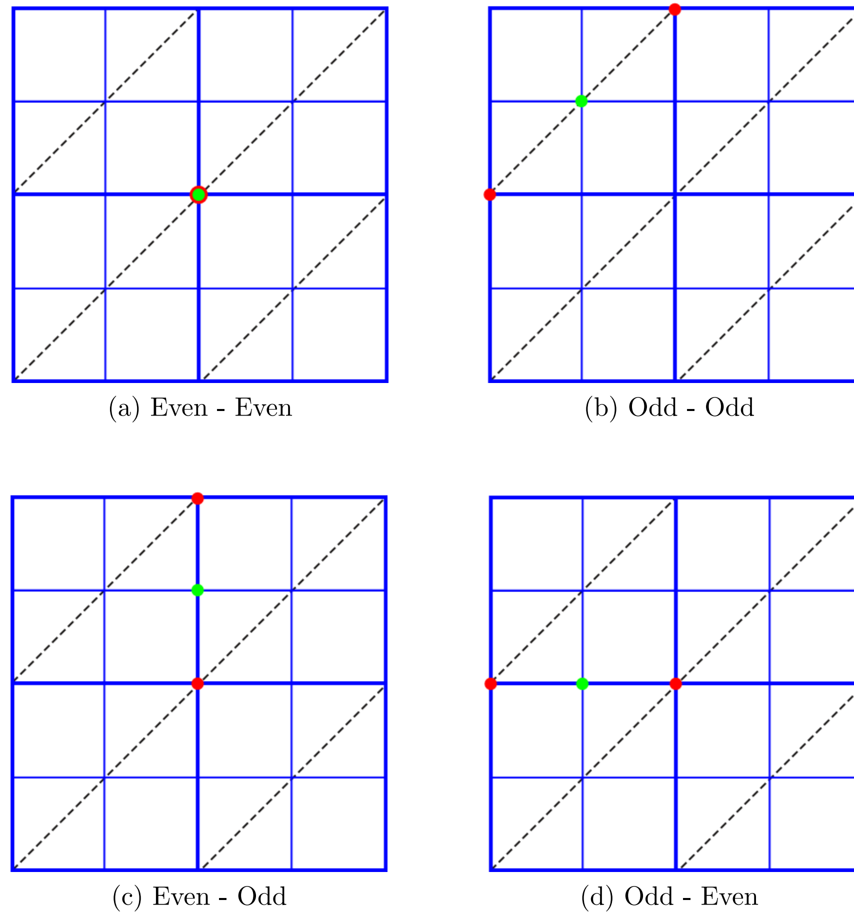


Figure 3.14: Demonstrates how the coarser level is sampled for each possible even/odd configuration. The dashed and bold lines represent the triangle edges in the coarser level. Note that in (a), the even – even sampling results in only one position being sampled, as this configuration matches the vertex in the coarser level.

Note that if a pixel position is odd-odd, we still only sample from two texels. This differs from the original GPU geometry clipmaps implementation. The reason for this is that while the vertex lies between four vertices in the coarser level, it falls on the edge of the two triangles which link these vertices. This edge only runs between two of the vertices, thus we only sample from these two vertices. Which two vertices we sample depends on the tessellation scheme chosen (i.e. how the adjacent vertices are linked with triangles in order to create the mesh).

3.4 Shadows

Shadows help to make virtual scenes appear more realistic, and are important as they provide visual queues to the position of object within a scene. The two most popular forms of shadow generation are shadow volumes[10] and shadow maps[54].

Shadow maps provide numerous advantages over shadow volumes. They allow for complex self-shadowing of objects, shadowing with alpha-tested geometry, and don't exhibit any complex problems with the camera position, unlike shadow volumes. They are also intuitively easier to implement than shadow volumes, which require complex stencil buffer usage.

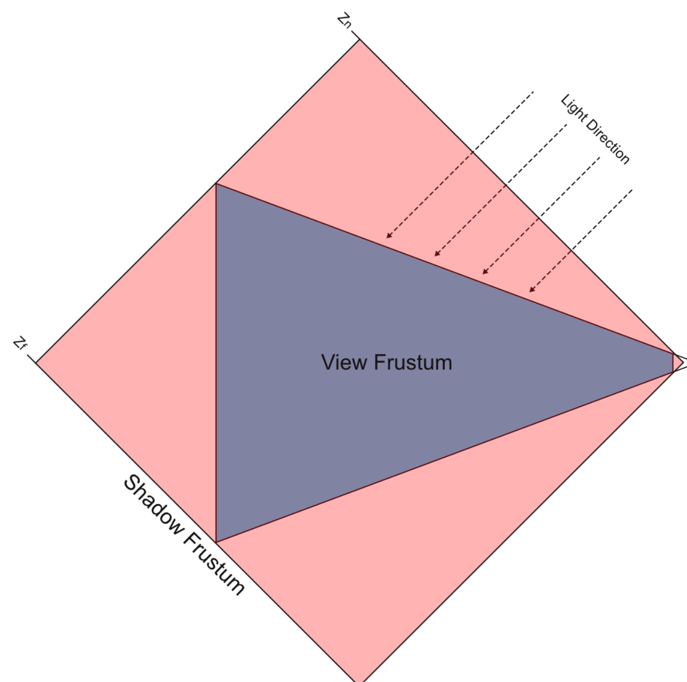
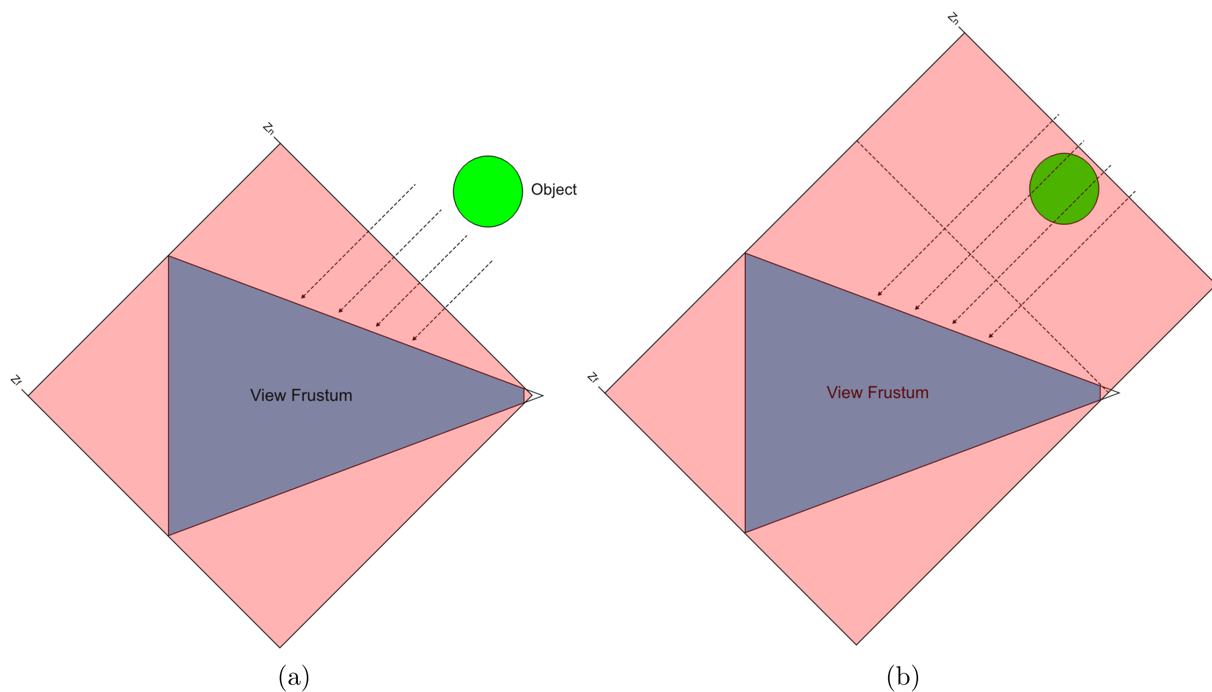


Figure 3.15: The camera set up for shadow mapping. An orthographic projection is used, facing in the lights direction. The lights view frustum (red) is made to cover the view frustum (blue).

In order to shadow a scene with shadow maps, the scene is rendered in two passes. First, the scene is rendered from the lights perspective. Global illumination sources, such as the sun, cause light to appear to come from the same direction, regardless of the position of the object within the scene. Thus an orthographic projection is used. This view volume

should cover any objects which may be seen from the cameras current viewpoint, as well as any objects outside the current view frustum, which could potentially throw shadows onto objects within the view frustum. Thus, if an object falls behind the near plane, the near plane should be moved backwards to cover this object. An example of this can be seen in Figure 3.16.The scene is then rendered, with the resulting depth texture (or shadow map as it is known), and the current model-view projection matrix being saved for later use. An example of a shadow map can be seen in Figure 3.17. Note that no rendering of colours should occur in this step.



(a) (b)

Figure 3.16: In (a) the camera is set up to render the shadow map. However, an object is behind the near plane, and thus should also be included in the shadow map. Thus, in (b), the near plane is moved backwards, in order to cover the object.

Next, the scene is rendered from the observers view point. When lighting the generated fragments, each fragment should be transformed, using the previously saved model-view projection matrix, into its corresponding position in the lights view frustum. The fragment then samples the saved depth texture, and compares its depth to that stored in the depth texture. If it is greater than the value stored in the depth texture, this means that there is another object between the fragment and the light, and thus should be shadowed.

Algorithm 5 C++ code snippet showing how the backplane is moved to cover any objects behind it.

```
//First we cull all the objects in the scene
m_objects.MoveFirst();
for( int i = 0; i < m_objects.GetNumItems(); i++ )
{
    //Note: Don't cull against the near plane...
    m_objects.GetCurrentNode()->SetMeshCulled( !Inside( 8,
        m_objects.GetCurrentNode()->GetBoundingBox(), 5, planes ) );
    m_objects.MoveNext();
}

//We then get the furthest position of an object behind the camera
m_objects.MoveFirst();
for( int i = 0; i < m_objects.GetNumItems(); i++ )
{
    if( !m_objects.GetCurrentNode()->GetMeshCulled() )
    {
        KThreeDVector* boundingBox =
            m_objects.GetCurrentNode()->GetBoundingBox();
        for( int j = 0; j < 8; j++ )
        {
            KThreeDVector position;
            position.m_x = (float) boundingBox[j].m_x;
            position.m_y = (float) boundingBox[j].m_y;
            position.m_z = (float) boundingBox[j].m_z;
            position = orthographicProjectionModelView * position;

            //and we extend the nearplane backwards to make sure that this
            //position falls within the view frustum
            if( position.m_z < minZ )
            {
                minZ = (float) position.m_z;
            }
        }
    }
    m_objects.MoveNext();
}
```

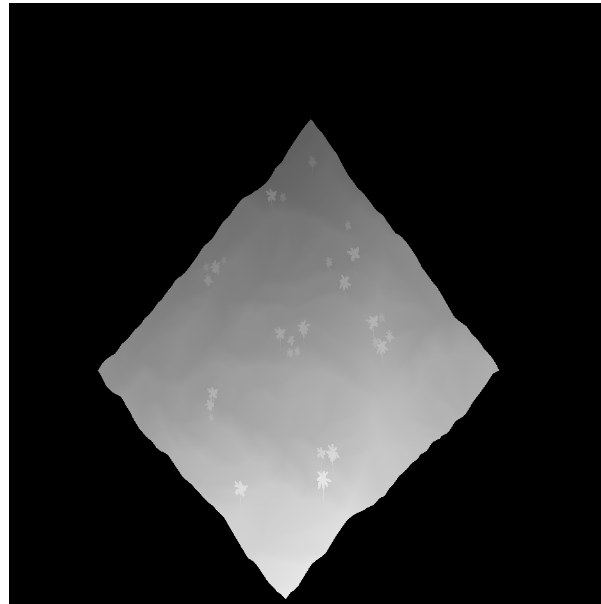


Figure 3.17: An example of a shadow mapping depth map. The terrain, as well as the palm trees on the terrain, can be clearly made out. Each fragment in the scene tests its position against this map, in order to check if it is shadowed.

Shadow maps are not without their problems. The shadows do not display soft edges, as in real life. There are a multitude of solutions to this problem. We chose a simple approach, where multiple pixels in the depth map are sampled by each fragment. The fragment is evaluated against each of these samples, and for each sample against which the depth test fails, the object appears slightly more in shadow. This is known as percentage-closer filtering (PCF). Our system uses a 4×4 PCF kernel, thus 16 different shadowed amounts are available. There are many more complex techniques available, such as exponential shadow maps. However, as shadowing is not the major focus of this thesis, and thus these are outside the scope of the thesis.

Secondly, due to rounding errors, erroneous self shadowing may occur. This is because the fragment may incorrectly fail the depth test, when it was the fragment that generated the depth value in the shadow map. By adding a slight offset to the value sampled from the depth map, this self shadowing may be eliminated.

The resolution of the shadow map effects the quality of the resulting shadows. If the shadow map resolution is too low, the resulting shadows appear blocky, as multiple fragments evaluate to the same pixel in the shadow map. However, for very large scenes, such as terrains, a single standard shadow map is not useful, as the resolution in the

foreground is too low. Using an extremely high resolution shadow map is wasteful, both due to the memory required, and because the extra detail is wasted on objects further away from the camera, where fewer fragments are generated.

Cascaded shadow maps[58, 12] provide an intuitive solution to this problem. Cascaded shadow maps divides the view frustum up into multiple sections. An example of this concept is shown in Figure 3.18. Each section is assigned it's own shadow map, which is cropped in order to cover only that frustum slice. Objects which fall within each frustum slice sample the corresponding shadow map to see if they should be shadowed. The frustum slices increase in size the further away from the observer they are. Thus, there in a decrease in shadow map resolution relative to the scene further away from the camera. This is ideal, as it corresponds to the reduction in generated fragments, and users are less likely to notice artefacts in the background than in the foreground.

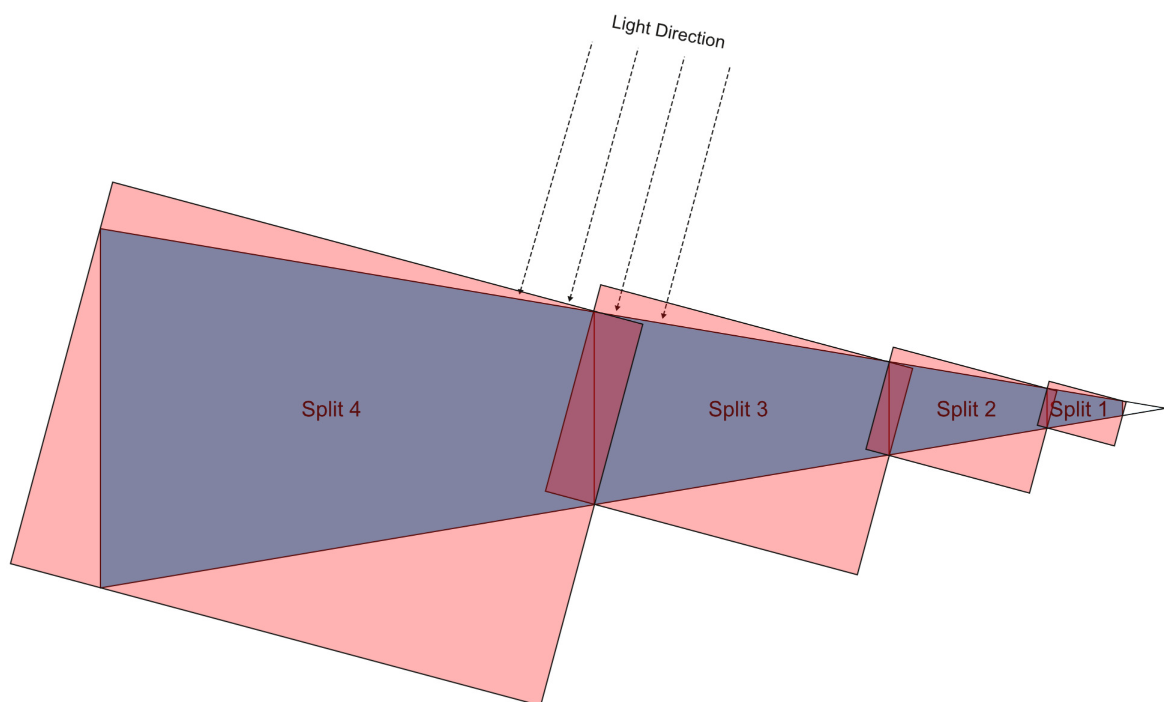


Figure 3.18: An example of the camera set up for cascaded shadow maps. The view frustum has been divided into 4 parts (labelled split 1-4), and a separate shadow map is fitted to each frustum split.

This solution is not without problems however. Firstly, there is a noticeable transition between shadows which fall across the division in frustum slices. In order to solve this, the view frustum slices are made to overlap slightly. Any fragments which are generated

in this region sample both of the corresponding shadow maps, and blend the results based on their position.

Secondly, as the view frustum moves and rotates, due to the bounding of the shadow map to the frustum slice, the resolution of the shadow map may change between frames, resulting in a shimmering effect. Also, as the relative position shadow map moves to match the view frustum, fragments in the resulting scene may evaluate to different pixels in the shadow map, resulting in a similar shimmering effect. We have implemented the method proposed by Valient[52], in order to solve these problems. The shadow map view frustums are limited to one size. This should be large enough to cover the view frustum slice, regardless of the angle. Additionally, the shadow map view frustum should only move in increments which correspond to the pixel size of the shadow map, causing the fragments to always sample effectively the same pixel in the shadow map. Combining these two techniques results in shimmering effects being eliminated.

3.5 Summary

This chapter presented our GPU geometry clipmaps implementation. The underlying theory, as well as the implementation specifics related to the technique were analysed. We observed how the terrain is rendered, and how the system updates the clipmap levels as the user moves around the terrain. Our simplified texturing algorithm was introduced, which may be used to texture the resulting terrain with very little additional overhead, which has been a problem with the GPU geometry clipmaps technique in the past.

GPU geometry clipmaps are capable of representing and rendering very large heightfield-based terrains in real-time. It uses very little bandwidth on the CPU-GPU bus, which increases overall rendering throughput. The resulting terrain is high quality, with no discernible artefacts visible at the borders of the clipmap levels. Additionally, we showed how the terrain may be shadowed with a cascaded shadow mapping approach. Due to the size of the terrain, multiple shadow maps are required in order to attain a sufficient shadow map resolution.

The performance of the GPU geometry clipmaps implementation is analysed in the results chapter.

Chapter 4

Particle-Based Terrain

The particle-based terrain system developed by Longmore et al.[29, 30] forms the basis of our particle-based terrain level of detail. The system is capable of simulating and rendering approximately 650,000 particles in real-time with a modern consumer GPU. Additional features have been added to the system, such as support for different particle sizes. This chapter analyses the components of the particle-based terrain system, and explores the new features and optimisations that have been made to the system.

Figure 4.1: Example of the particle-based granular terrain simulation, interacting with dynamic objects.

The system simulates the natural interactions of sand using rigid bodies, which are made up of groups of four particles, in a tetrahedral structure. The particles then interact with particles from other rigid bodies, which applies a force to the rigid body. The system produces realistic particle-based terrain, which exhibits physically correct interactions. The particle system leverages the computational power of modern GPUs, while remaining hardware agnostic. This allows us to target a wide variety of hardware.

4.1 Particles and Rigid Bodies

Previous particle-based simulations used spherical particles in order to simulate granular material. However, the use of spherical particles requires complex algorithms in order to simulate friction. Friction is required within the system in order to allow for sand pile formation. Bell et al.[5] proposed the use of a multiparticle system, where particles are grouped together to form rigid bodies. Each rigid body is made up of four particles, which form a tetrahedron. This is advantageous, as it allows the rigid bodies to interlock, thus introducing friction to the system, as well as producing physically accurate granular interactions.

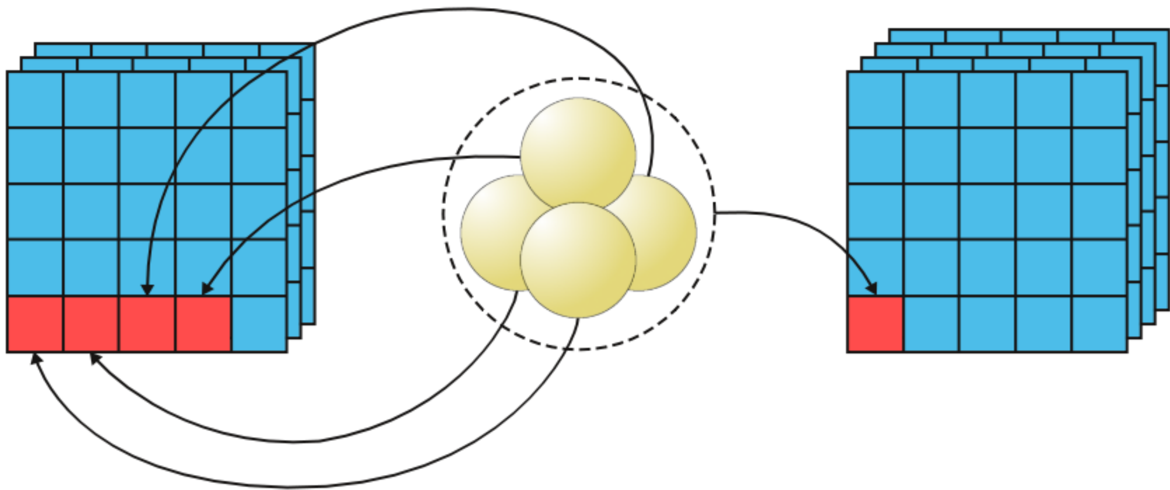


Figure 4.2: Shows how the granules are made up of four different particles, and shows how the particles and rigid body map to the storage textures[30].

All the data required for the particles and rigid bodies is stored in textures. Textures

may be used as a data source, or a data target, and thus present an attractive method to store this information. Additionally, texture caching allows extremely fast access to data which is accessed in spatially local area. Each texture is a four channel floating point texture, and represents a property of the group of particles or rigid bodies. These properties are position, orientation, momentum, and angular momentum for the rigid bodies, and position, momentum, force and offset for the particles.

The particles are linked to the rigid bodies using unique identifiers. Each particle is allocated an identifier, based on its position within the texture. This identifier is stored in the alpha channel of the position texture. The rigid body then stores the identifier of its first constituent particle. As each rigid body is made up of a known number of particles, it is then simple to retrieve the other constituent particles.

The number of particles and rigid bodies in the simulation is limited by the size of the textures. If more particles are added than the current limitation, new textures need to be created. The current data is then copied to the new texture, and then additional particles and rigid bodies may be added. This operation is obviously extremely expensive, and should thus be avoided wherever possible. The framework presented assumes a constant volume of sand, i.e. sand is added at the start of the simulation, and sand is never added or removed from the system.

4.2 Updates

The system maintains two textures for each property that is stored for the particles/rigid bodies. Each frame, one texture acts as the data source, and one texture acts as the data target. The data is processed by the GPU, and the results are written to the target texture, which then becomes the source texture for the following frame. The advantage here is that because the result is written to a different texture, the source data remains intact, so the result for each particle is based on a constant, non-varying set of data.

The system uses the fragment shader to process the updates, by using render to texture techniques. This is different to how GPU computations are usually processed. GPU computation libraries, such as CUDA[38] or OpenCL[17] are generally used, instead of the fragment shader, as they allow for more generalised computations to be performed.

However, this approach is hardware agnostic, as it uses the OpenGL library, which has excellent support from GPU vendors, to perform the rendering, and thus the computations. CUDA, meanwhile, is nVidia specific, and OpenCL has poor driver implementations from some vendors.

The updates are processed using the following steps:

- Map particles to 3D Grid
- Update particle forces based on collisions with adjacent particles
- Update rigid bodies based on the forces of the constituent particles
- Update particle positions based on the position and rotation of the rigid bodies

One issue which needs to be addressed is the choice of time step. A common feature amongst particle simulations is that the equations used are sensitive to the simulation delta time. If the simulation is sampled too finely or coarsely, i.e. the time step is too small or too high, the equations breakdown, and produce poor results. Thus we have chosen a fixed time step for our simulation. Unfortunately, this means that the simulation may appear to run slower or quicker depending on the power of the hardware used, as well as the computational complexity of the particle system (i.e. how many particles are being simulated). As our work is a proof of concept, the choice of a fixed delta time has provided acceptable results.

4.2.1 Mapping of particles to 3D Grid

In order to detect collisions between the various particles in the simulation, a texture representing a 3D grid is created. This 3D grid represents the volume in which the particles and rigid bodies exist, effectively discretising the space into a voxelised format. The ID of each particle is added to the grid node which corresponds to it's position in 3-dimensional space. As each texel within the texture is composed of four values, a maximum of four particles may translate to a single grid node. One of the side-effects of this system is that particles are allowed to overlap slightly before collisions are detected. This is known as a soft-body dynamic system.

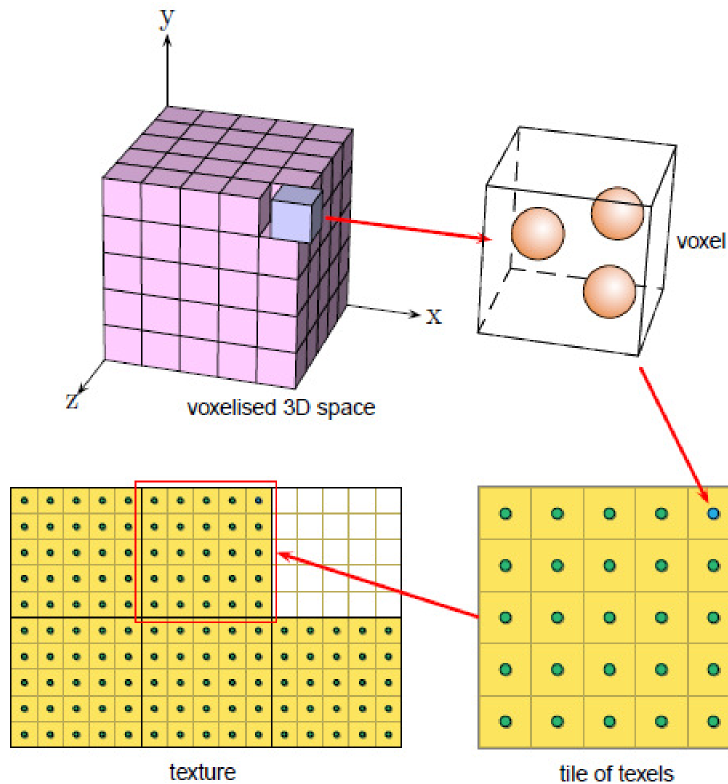


Figure 4.3: The 3D grid is represented by a 2D texture. Each slice in the 3D grid is stored as a tile within the 2D texture. Each texel represents a voxel within the 3D grid, and can store up to 4 particles (one per colour channel). This diagram, from Longmore[29], shows the layout for a $5 \times 5 \times 5$ voxel area of space.

Instead of using a true 3D texture, we use a large 2D texture. This texture is arranged to represent side-by-side texture layers, effectively creating a 3D texture. This allows us to use non-power of two domain sizes. Also, traditionally when rendering to a 3D texture, a separate render pass is required for each texture layer. When a 3D texture is flattened into a 2D texture, this is no longer a concern.

Particles are mapped to this 3D grid using a rendering based technique. A fragment is generated for each particle; the position of which corresponds to the position of the particle in the simulation. This position is calculated as in Algorithm 7. However, as multiple particles may map to the same texel within the 3D grid texture, the results from these particles will overwrite each other. Harada [20] created a method to deal with this problem. Four rendering passes are used to write the data to the 3D grid. By using

different colour masks, and different stencil and depth tests for each pass, different particle IDs are written to each channel in the texture.

Colour masks allow only the specified colour channel to be written to the rendering output. Each fragment is designated a depth equal to its ID, and writes its ID as the colour for each channel. The rendering order of the fragments is known, as the vertices are passed to the vertex shader in particle ID order. The four passes are rendered in the following manner:

First Pass Stencil tests are disabled and a red colour mask is used. The depth buffer is cleared. Depth testing is enabled and passes smaller depth values. Thus, the particle with the lowest ID is written to the red colour channel.

Second Pass Stencil tests are enabled, and set to increment the stencil value whenever a fragment is written. A fragment will fail the stencil test if a value has already been written to that fragment position this pass (i.e. stencil value > 0). Depth testing is enabled, and set to pass larger depth values. A green colour mask is used. Thus, the first fragment to pass the depth pass (i.e. the second lowest particle ID) will have it's ID written to the green colour channel.

Third Pass The stencil buffer is cleared, and the same stencil and depth tests are used as in the second pass. A blue colour mask is used. Thus, the fragment with the third lowest particle ID is written to the blue channel, as the first two fail the depth test, and the last one fails the stencil test.

Fourth Pass This pass is the same as the third pass, except an alpha colour mask is used. Thus, the final particle ID is written to the alpha channel.

As we can see, this method operates by using a stencil test to make sure that only one fragment is written each pass. The first fragment to pass the depth test each pass is thus written to the corresponding colour channel, and blocks further fragments from being rendered to that location. Each pass a different colour mask is used. Thus each pass renders the next largest particle ID that maps to that location to the corresponding colour channel. Note that the depth buffer is only cleared. Also, as mentioned earlier, the grid is limited to four particles per texel (due to the available colour channels), and thus only four passes are required. Algorithm 6 shows the C++ code used for this method.

Algorithm 6 C++ code snippet showing the OpenGL code used to map the particles to the 3D grid each frame.

```
//pass 1
glEnable( GL_DEPTH_TEST );
glDepthFunc( GL_LEQUAL );
glClearDepth( 1.0 );
glClear( GL_DEPTH_BUFFER_BIT );
glColorMask( GL_TRUE, GL_FALSE, GL_FALSE, GL_FALSE );

_draw( coords );

//pass 2
glDepthFunc( GL_GREATER );
glColorMask( GL_FALSE, GL_TRUE, GL_FALSE, GL_FALSE );

glEnable( GL_STENCIL_TEST );
glStencilOp( GL_KEEP, GL_KEEP, GL_INCR );
glStencilFunc( GL_GREATER, 1U, 0x11111111U );
glClearStencil( 0 );
glStencilMask( 0x11111111U );
glClear( GL_STENCIL_BUFFER_BIT );

_draw( coords );

//pass 3
glClear( GL_STENCIL_BUFFER_BIT );
glColorMask( GL_FALSE, GL_FALSE, GL_TRUE, GL_FALSE );

_draw( coords );

//pass 4
glClear( GL_STENCIL_BUFFER_BIT );
glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_TRUE );

_draw( coords );
```

4.2.2 Update particle forces

In order to calculate the forces applied to each particle, the first step is to find the texel within the 3D texture, which corresponds to the particle. The size of the simulation domain, the position of the particle and the dimensions of the texture are all known, and thus it is easy to calculate the texture coordinates of the particle within the 3D grid. The following algorithm performs this calculation:

Algorithm 7 Pseudocode to calculate texture coordinate for a particle within the 3D grid texture, corresponding to the position of the particle.

```
function GenerateTextureCoordinate(position)
    x ← floor(position.x/domainSize ∗ tileSize)
    y ← floor(position.y/domainSize ∗ tileSize)
    z ← floor(position.z/domainSize ∗ numTiles)
    x ← x + mod(z, numTilesPerRow) ∗ tileSize
    y ← y + floor(z/numTilesPerRow) ∗ tileSize
    x ← x/textureSize
    y ← y/textureSize
    return x, y
```

Once the particle has been located, the system checks each adjacent grid node (including its own grid node). This requires 27 texture lookups (i.e. a $3 \times 3 \times 3$ cube). If a particle from another rigid body is found in an adjacent node, we then perform a collision between the particles. The resultant force of the collision is calculated using the formulas from Bell et al.[5]:

$$F_n = -\gamma_n \xi_n^{\frac{1}{2}} \dot{\xi}_n - k_n \xi_n^{\frac{3}{2}}$$

$$\mathbf{F}_s = -min(\mu F_n, \gamma_s \|\acute{\mathbf{v}}_s\|) \frac{\acute{\mathbf{v}}_s}{\|\acute{\mathbf{v}}_s\|}$$

where $\acute{\mathbf{v}}_s$ is the velocity in the shearing direction, and ξ_n is the penetration depth. This is repeated for each adjacent node, and the forces are summed to find the final force acting upon this particle, i.e.

$$\mathbf{F}_p = \sum_{i \in P - \{p\}} \mathbf{F}_i$$

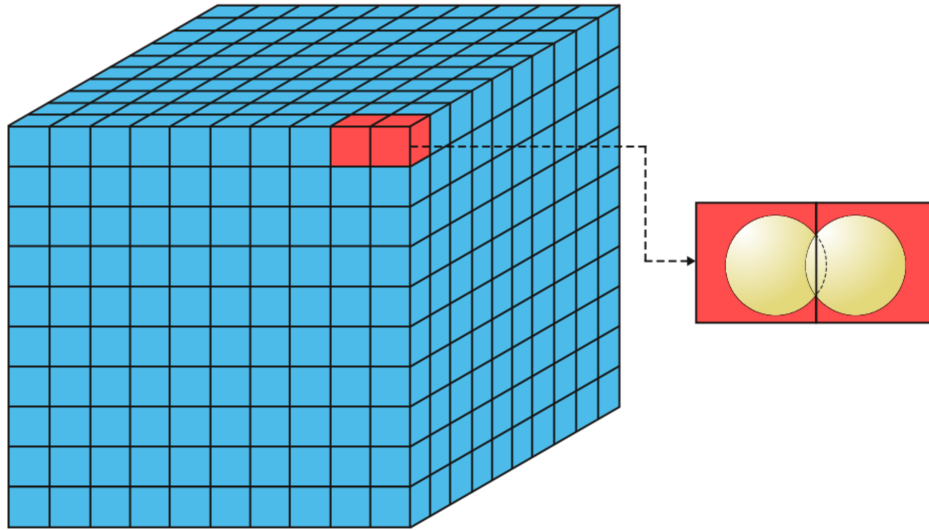


Figure 4.4: Shows how collisions are resolved using the 3D Grid. Two particles in adjacent voxels are intersecting, and thus a collision occurs.

Additionally, each particle is affected by gravity. This force needs to be added to the resulting force for the particle. The total force experienced by a particle each time step is then represented by the equation:

$$\mathbf{F}_p = \sum_{i \in P - \{p\}} \mathbf{F}_i + m\mathbf{g}$$

where m is the mass of the particle.

4.2.3 Update rigid bodies

Once the forces on the particles have been calculated, the linear force experienced by the rigid body is calculated as:

$$\mathbf{F}_{rb} = \sum_{i=1}^{j} \mathbf{F}_i$$

i.e. we sum the forces experienced by the particle constituents of the rigid body in order to determine the final force experienced by the rigid body. Additionally, we can calculate

the torque experienced by the rigid body, by calculating the torque applied by each force at the particle offset, i.e.

$$\mathbf{T}_{rb} = \sum_{i=1}^{j} \mathbf{r}_i \times \mathbf{F}_i$$

where $\mathbf{r}_i$ is the position of the particle in respect to the rigid bodies center. The orientation of the rigid body is then updated using this torque value, and the resulting orientation is stored as a quaternion in the orientation texture. The corresponding GLSL shader code is shown in Algorithm 8.

Algorithm 8 GLSL shader code snippet showing how the forces are applied to the rigid body updates.

```
vec2 id_to_texCoord( int id )
{
   vec2 v;
   float cid = float( id );
   v.y = floor( cid / particleTexWidth );
   v.x = ( cid - particleTexWidth * v.y );
   return v;
}

//.....

vec3 totalForce = vec3( 0 );
vec3 torque    = vec3( 0 );

for( int i = 0; i < 4; ++i )
{
   vec3 rpos = texture2DRect( relPositionTexture, id_to_texCoord(
      firstParticleId + i ) ).xyz * cellDiameter;
   vec4 force = texture2DRect( forceTexture, id_to_texCoord( firstParticleId +
      i ) );

   totalForce += force.xyz;
   torque     += cross( rpos, force.xyz );
}
```

4.2.4 Update particle positions

Once the rigid bodies have been updated, the particle positions must be updated to reflect the new position and rotation of the parent rigid bodies. As each particle contains an offset from the rigid body, the new position of the particle can easily be computed using the formula

$$\mathbf{x}_i = \mathbf{x}_g + R_g \mathbf{r}_i$$

where $\mathbf{x}_g$ and R_g are the position and rotation of the rigid body, and $\mathbf{r}_i$ is the offset of the particle from the rigid body. Additionally, the velocity of the particle is calculated using the following formula:

$$\mathbf{v}_i = \mathbf{v}_g + \omega_g \times \mathbf{r}_i$$

where $\mathbf{v}_g$ and ω_g are the velocity and angular velocity of the rigid body.

4.3 Containing the Particle System

In order to form a volume of sand, the particles in the simulation need to be constrained inside a container. Without a container to hold the particles, they would simply fall through space (under the force of gravity). Various options are available, such as spheres, cylinders or boxes.

We chose to use a box to contain the particle system. The reasons for this is that firstly, it is relatively simple to perform the calculations for a box. An open topped box may be represented using 5 planes. A plane may be represented by the formula:

$$Ax + By + Cz + D = 0$$

where A, B and C are the x,y and z components of the planes normal N, and D is the distance from the origin. Using this plane representation makes performing collisions with a plane relatively inexpensive, as the distance of the object to the plane may be calculated using the following formula:

$$distance = \mathbf{N} \cdot \mathbf{P} + D$$

where P is the position of the particle which is being evaluated. If the position is less than

the particle radius, the collision is performed using the method described in the previous section.

The second, and more important reason for the choice of a box, is that a box has a square, or rectangular shape. This makes mapping the particles from a heightfield-based representation, or converting to a heightfield-based representation easier, as the box corresponds to an area in the heightfield. More information on the conversions can be found in Chapter 5.

4.4 Rendering

As explained earlier, a particle system may contain hundreds of thousands of particles. All of these particles must be rendered each frame, which is extremely costly. Regular particle rendering techniques, such as textured billboarded quads, are far too expensive, due to the number of draw calls and vertices which need to be processed. Thus, an alternative approach to rendering particles is required.

A splatting technique[53] is employed to render the particles. Splatting is a technique which renders objects using points. This is usually used to render volumetric objects, but has been adapted to render objects with a large number of vertices[44], such as 3D scanned objects. Points of various shapes may be used, such as squares or circles.

The particle system displays many characteristics which make it an ideal candidate for splatting. Firstly, the particles in the simulation are spherical, and so may be accurately represented using circular discs. The sand particles are assumed to be opaque, and thus do not need to be sorted according to depth. Additionally, the choice of a splatting technique also holds many inherit advantages to rendering particle systems of this kind. Each particle only requires a single vertex to be processed, which reduces the vertex computations required to render the particle system. Splats do not need to be oriented in screen space, unlike billboarded primitives, which further reduces the rendering overhead. Also, splats are rendered using simple points, which are extremely quick to render.

In order to perform the rendering, a vertex buffer object (VBO) is generated. One vertex is added to the VBO for each particle within the simulation. Its position is the

texture coordinate which corresponds to the particle within the particle position texture. The vertices are then rendered using the `glDrawArrays` call, with the rendering mode set to `GL_POINTS`. The vertex shader reads the position of the particle from the position texture, and sets the position of the resulting point to that position. The points produced by `GL_POINTS` are square, and thus need to be trimmed by the fragment shader. Any fragments which lie outside of a circle are simply discarded. This method allows the entire particle system to be rendered in one call.

An example of a rendered particle system can be seen in Figure 4.1. As shown in Chapter 6, rendering the particle system is relatively cheap when compared to the update procedure. However, it still remains relatively expensive to render the particle system in comparison to other objects, due to the number of vertices processed, and the number of fragments generated by hundreds of thousands of particles.

Another downside to this technique is that each of the particles appears spherical, thus giving the particle system a uniform look, even when different textures are applied to the particles. One solution to this problem could be to use instanced geometry, thus giving the particles a range of different looks. This, however, has been left as future work.

4.4.1 Shadows

In order to shadow the particle system, we employ a shadow mapping technique, as in Section 3.4. See Figure 4.5 for an example of a depth map obtained from rendering the particle system with shadow mapping enabled. Unlike the terrain, the particle system only covers a small area of the scene. Additionally, as has already been noted, rendering the particle system is relatively expensive when compared to other objects in the scene. Thus, rendering the particle system as part of the cascaded shadow maps technique from Section 3.4 is not ideal, as the particle systems would possibly be rendered in multiple shadow maps.

Thus, each particle system is assigned a single shadow map, which covers that particle system. As the particle system only covers a small area of the terrain, sufficient resolution is obtained using only a single shadow map. Additionally, due to the high resolution of the shadow map in comparison to the small area of the terrain covered, no PCF filtering is

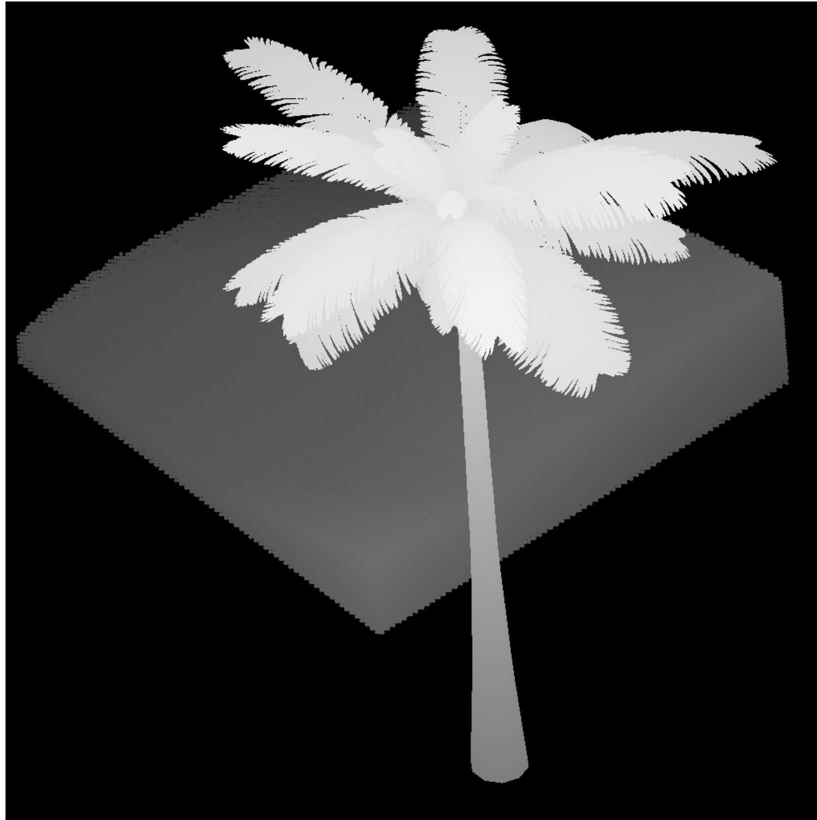


Figure 4.5: An example of the depth map generated during rendering of the particle system with shadow mapping enabled.

required, which reduces the number of texture accesses required by the fragment shader. Hundreds of thousands of particles may be rendered, with each particle producing multiple fragments. Thus, the number of texture accesses is extremely high, and reducing the number of texture accesses by disabling PCF filtering is thus advantageous.

Similar to the shadow mapping of the heightfield-based terrain, the lights view frustum is clipped to the bounds of the terrain section. Once the view frustum has been fixed to the bounds of the particle system, the rear plane is moved backwards, to cover any objects behind the camera. As the particle system remains in one place, the lights view frustum needn't be updated each frame. This also means that we need not address the problem of shadow shimmering, as the fixed position and size of the light view frustum ensures that this problem does not arise.

As the particle system is rendered using points, there is only one vertex per particle. Thus, the position of the particle in shadow space needs to be calculated per fragment,

using the fragment position, as no other vertices exist amongst which to interpolate the position in shadow space. See Algorithm 9 below for how this position is calculated from the fragments position in window space.

Algorithm 9 GLSL shader code to calculate a fragments position in shadow space.

```
//Get the position of the fragment in window space
vec4 shadowPosition = gl_FragCoord;
//Convert from window space to screen space
shadowPosition.x = shadowPosition.x / windowWidth;
shadowPosition.y = shadowPosition.y / windowHeight;
shadowPosition = shadowPosition * 2.0 - 1.0;
shadowPosition.w = 1.0;
//Transform to world space
shadowPosition = inverse( gl_ModelViewProjectionMatrix ) * shadowPosition;
//Transform to shadow space
shadowPosition = ShadowMVP * ( shadowPosition / shadowPosition.w );
shadowPosition = shadowPosition / shadowPosition.w;
//Convert to texture space
shadowPosition.xyz = shadowPosition.xyz * 0.5 + 0.5;

//Test the depth against the shadow map
float distanceFromLight = texture2D( ShadowSampler, shadowPosition.xy ).z +
    bias;
float shadow = distanceFromLight < shadowPosition.z ? 0.5 : 1.0 ;
```

4.5 Scaling of Particles

One of the features of the LOD system is the ability to use coarser particle simulations further away from the observer, and finer grained, more detailed particle simulations closer to the observer. However, Longmore's particle system lacks the ability to scale the particle sizes. Thus, the particle system had to be adapted to support particles of different sizes.

We tried many different methods to achieve this. The first method was to only perform collisions when the particles get closer to each other, by using a higher resolution grid,

with smaller particles. As the particles are closer to each other when the collisions are computed, the forces resulting from the collisions also need to be scaled. However, this scaling is not linear, requiring constants to be tuned for each particle size. This method is further complicated by other scaling within the particle system, that is used to address rounding error due to single precision floating point values.

Next we tried a similar approach. As in the previous approach, a denser 3D grid was used, with smaller particles. Once again, greater forces are generated by the collisions between the particles. However, in this approach we simply scale the movement of the position of the rigid body, and the resulting imparted angular momentum. However, this scaling also turned out to be non-linear, and despite attempting to tune the parameters, resulted in unstable particle behaviour.

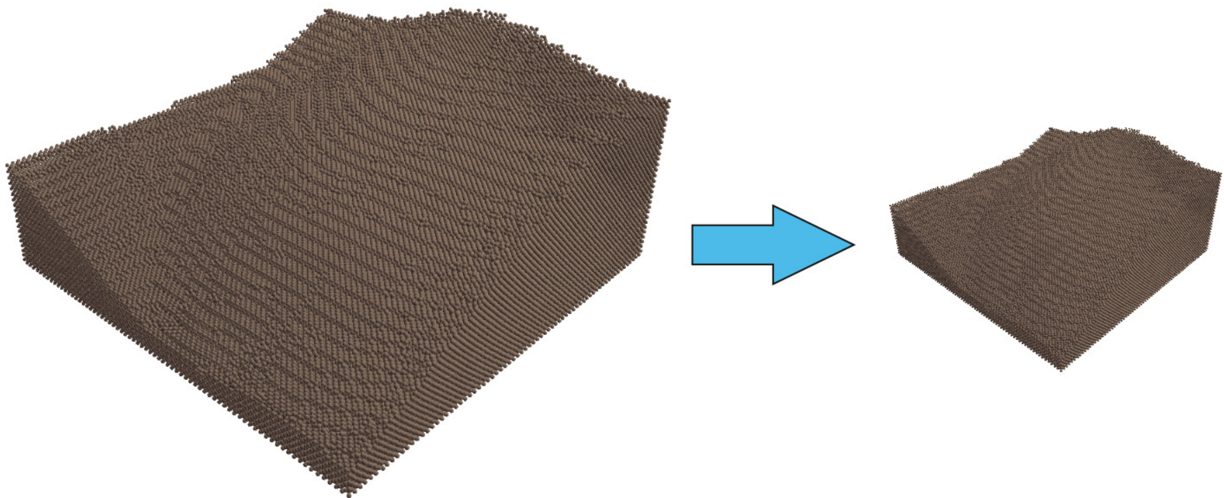


Figure 4.6: Shows how the particle system is scaled, by simply rendering it at a smaller scale, whilst processing updates at a set scale.

The final method we tried is much simpler. The updates and collisions are processed with particles of the regular size. The renderer then scales the particle sizes and positions, creating the appearance of a smaller particle size. This concept is illustrated in Figure 4.6. The velocity of the objects needn't be scaled, as this is already scaled by the scaling of the position. Only the initial velocity of a new object entering the system must be scaled, as it will be moving faster in relation to the objects than in the virtual world. The only force that needs to be scaled within the particle system then is that of gravity, as the movement of the particles in response to other forces is automatically scaled due to this

representation. Gravity must be scaled, as the distance covered by a falling object in the virtual world should remain constant, regardless of the scale of the particle system. The position of an object can be expressed as: $\mathbf{pos} = \mathbf{pos}_0 + \mathbf{v}t + \mathbf{a}t^2$. But since the force of gravity results in a constant acceleration, this can be scaled linearly. Thus, no arbitrary parameter tuning is required when using this method. This method also resulted in more stable particle behaviour than the previous methods.

4.6 Interactions with Models

In order for the particle system to interact with a model, the model must be converted to a particle-based representation. This conversion is detailed in Section 5.5, as it is handled by the terrain manager in our implementation. This particle-based representation corresponds to the shape of the model. Each model is made up of a single rigid body, and many particles. The model's particles and rigid body are stored in the same textures as those of the terrain granules. Thus, collisions between the model and the terrain occur in the same way as with the granules. Collisions with other models within the system may also occur this way.

As each model is made up of a single rigid body, and many particles, we thus have to sum the forces imparted on each particle within the model in order to calculate the force exerted on the model by interactions with the terrain or other objects. In order for this to occur, the fragment shaders, which process the updates, need to know how many particles the model consists of, and the IDs of those particles. Additionally, to allow the correct movement and response of the object to these forces, the mass of the object should be passed to the shaders. All this information is passed to the shaders using uniform arrays, so they can use this information to calculate the correct interactions for the model. Each entry within each array contains the information for a separate object. Thus multiple objects may exist within a particle-based simulation.

Although the particle-based representation should interact with the terrain, the actual model should still be rendered in its original form. The particle-based representation should not be rendered. In order to achieve this, the first particle ID of the first particle-based model added to the particle system is stored. We only render particles with IDs up to (not including) this particle. Each frame, the positions and orientations of the models are read

from the rigid body textures, using `glReadPixels`. The actual models are updated to this position and rotation, and then rendered in their original form. This causes it to seem as though the actual model is colliding with the terrain, when, in fact, a particle-based representation of the model is responsible with these collisions.

4.7 Summary

In this chapter, the particle-based granular terrain simulation of Longmore et al.[30] was explored. This system forms the basis of our particle-based terrain level of detail system. The relationship between particles and granules was observed, with each granule consisting of four particles in a tetrahedral structure, in order to allow the interlocking of granules, thus producing physically correct particular interactions. We saw how the various attributes pertaining to the particles and rigid bodies are stored in textures, which allows for a rendering-based update mechanism. This update mechanism was examined, and it was shown how, by mapping the particles to a grid, collisions may be efficiently computed. Furthermore, the equations for calculating the forces applied to the rigid bodies through collisions were shown, along with the equations for computing the updates to the particle and rigid body positions.

The rendering of the particle system was also addressed. It was shown that a splat-based rendering system can be used to efficiently render the entire particle system. This splat-based approach takes advantage of the spherical particle representation to render the particles as individual points. By increasing the size of the points to match the particle size, and by applying some simple shading to the points, the rendered points mimic the appearance of spheres.

The scaling of the particle system was addressed. We explained how the particle system may be easily be scaled by simply scaling the rendering of the particles. In this way, the particle system may be efficiently scaled without the need for arbitrary parameter tuning. Finally, we addressed the collisions of arbitrary models with the system. We briefly described how this is achieved. Section 5.5 in the following chapter details how models are converted to a particle-based representation, so that they may interact with the terrain.

Chapter 5

Terrain Manager

At the core of the level of detail (LOD) technique lies the terrain manager. This component is responsible for switching between the heightfield-based terrain and the particle-based, in both directions, and converting models to a particle-based representation, so that they may interact with the terrain. It also holds pre-initialised copies of particle-based terrain system at different scales, and manages which of these particle systems are active. This ensures that new areas of particle-based terrain can be quickly introduced to the system, without the costly set up. Figure 5.2 shows the broad architecture of our system.

Figure 5.1: An example of a terrain with a particle system and dynamic objects interacting with it.

This chapter presents each of the individual components of the particle-based terrain manager. We show how the terrain manager handles multiple particle systems and how it is able to convert between the different terrain representations. Additionally, the conversion of models to a particle-based representation is presented. This is important, as it allows models to interact with the particle-based granular terrain simulation.

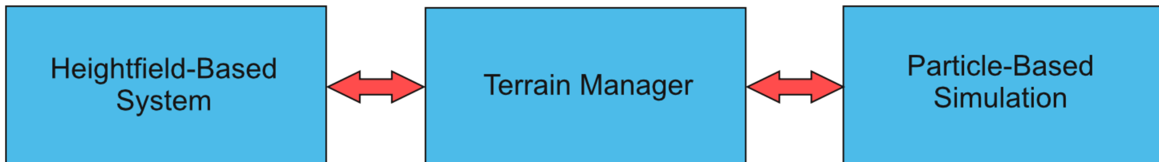


Figure 5.2: The terrain manager architecture. It is responsible for performing the transitions between the two terrain systems, and passing information between them. Additionally, it is responsible for converting models to a particle-based representation.

5.1 Management of Particle-Based Terrain Systems

The particle-based granular terrain simulation has an expensive set up. This includes initialising the textures to hold the particle and rigid body attributes, setting up the frame buffer objects, loading the shaders, and instantiating the renderer. In order to avoid the overhead of instantiating particle systems at runtime, the particle-based terrain manager maintains a set of pre-instantiated particle systems. When a new particle system is required to simulate the interactions between a dynamic object and the terrain, the particle system manager simply populates one of the inactive particle systems, and activates it to simulate the interactions. This avoids the overhead of instantiating a new particle system during the simulation run-time.

In section 4.5, it was shown that the particle-based simulation may be scaled, in order to create finer grained particle simulations. As the distance between the observer and the particle system increases, particle systems of different scales can be utilised. Finer grain, computationally expensive particle systems are required closer to the observer, in order to maintain the high fidelity of the simulations. However, particle systems which lie further away from the observer need not be as high fidelity, as the observer is unable to see such fine grain detail at these distances. Thus, the terrain manager system maintains multiple particle systems of different scales. Three different particle system scales are used. The

scale of a particle system is set at initialisation. If we add additional particle scales, then we either have to reassign particle systems from the other scales to the newly added scale, or we have to increase the total number of particle systems. Both of these options are undesirable, as the former may lead to us running out of particle systems for a particular scale, whilst the latter increases the memory usage of the system. Beyond this, we thought that breaking the terrain into close, medium and far distances made intuitive sense.

When a new particle system is required to simulate dynamic interactions with the terrain, an inactive particle system of the corresponding scale is selected. The scale of the particle system is selected based on the position of the interaction with regards to the observer. Figure 5.3 shows the various regions on a terrain, which correspond to the various particle system scales.

Assuming that interactions may take place at any point on the terrain, fewer fine grained simulations are required, as the area covered which corresponds to the fine grained simulations is much smaller than the area covered corresponding to the coarser simulations. Coarser simulations contain fewer particles, which means that we can run more of these simulations at the same time. Thus, our approach allows for large areas of terrain to be simulated using the particle-based system.

The disadvantage of this approach is that the textures for the particle systems (see Section 4.1) remain resident in memory while the particle system is inactive. Thus, there is a limit to the number of particle systems which may be used. However, as there are more coarser simulations than fine-grain simulations, the memory usage of the system is somewhat constrained, as coarser simulations use less memory than finer-grain simulations. Another disadvantage is that the particle-based simulations have a set scale. As the observer moves closer to a coarser simulation, the simulation remains at that scale, resulting in the coarser approximation becoming apparent to the user. In order to solve this, the particle system would have to be redesigned to allow for a dynamic shifting scale according to the distance of the simulation from the observer. This is beyond the scope of this thesis, and has thus been left as future work.

When active, particle simulations incur a very high computational cost. This suggest that the system can only support a small number of active simulations at any one time. However, dynamic objects tend to come to rest after interacting with the terrain. Once this occurs, all the additional computational resources which are being used to simulate particle

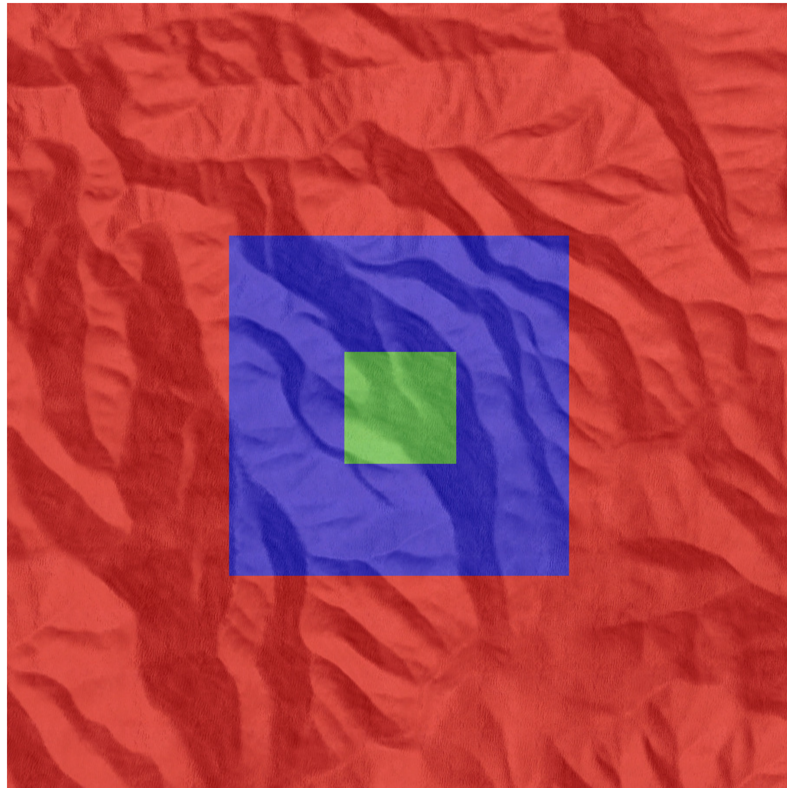


Figure 5.3: Illustrates the division of terrain for the different particle scales, based on the position of the observer. Coarse particle systems are used for the red area, finer systems for the blue area, and still finer systems for the green area.

system are effectively wasted, as they make no changes to the terrain. To counteract this, the simulations assume that if all the dynamic objects in the simulation remain stationary for more than two seconds, that the system has come to a rest state. The system then pauses the simulation, leaving all particles in their current position. The particle system is still rendered each frame, creating the illusion that there are more active particle systems than there actually are. This is effective, as rendering the particle system is considerably cheaper than processing the dynamic updates each frame [29]. If the dynamic object begins moving again, or a new dynamic object enters the simulation, the dynamic updates resume. However, if no changes occur, the system will be converted back to a heightfield representation after 30 seconds. This was chosen as a reasonable time frame within which we may expect new objects to be added to the system. The shorter this time frame, the more likely it is that an object will be added to an area after it has been converted back to the heightfield-based representation. On the other hand, if this period is too long, we could run out of particle systems with which to simulate new interactions with the terrain.

5.2 Rendering

In order to render both the particle-based terrain and the heightfield-based terrain at the same time, the heightfield-based terrain needs to know the position of the particle-based terrain simulations. This is to prevent both systems displaying the same portion of terrain at the same time, which will result in artefacts, especially as the heightfield-based terrain doesn't deform in real-time, as the partical-based terrain does.

In order to do this, the terrain manager passes the position and sizes of the particle systems to the heightfield-based system every time a new system is activated or a previously active system is deactivated. The heightfield-based system then don't render the areas of terrain which correspond to active particle systems, except for an alpha-blending region around the edges. These alpha-blending regions help to smooth the transitions between the heightfield-based system and the particle-based system – see Figure 5.1. Likewise, the particle-based simulation is also alpha-blended around the edges. Furthermore, when the heightfield-based terrain is converted to a particle-based system, and back again, the two systems must be alpha blended between, to prevent a "popping" effect. This is discussed in Section 5.3.

5.3 Conversion from Heightfield to Particle System

In order to convert from a heightfield to the particle-based terrain representation, the system needs to create a volume of particles. The height of the volume at each point on the x-z plane must correspond to the height stored in the heightfield. Thus, the first step in the conversion is dividing up the area covered by the particle system into a grid. The size of each unit in the grid corresponds to the size of one of the particle-based granules in the particle system. The heightfield is then sampled at each point on this grid. Bilinear interpolation[27] is used to infer the height at points which fall between texels in the heightfield. This height is used to deduce the number of rigid bodies which must be stacked at that point in order to reach that height on the terrain.

Arrays to store the rigid body and particle attributes are created. The grid is iterated over, and at each point on the grid, the positions of rigid bodies and their corresponding

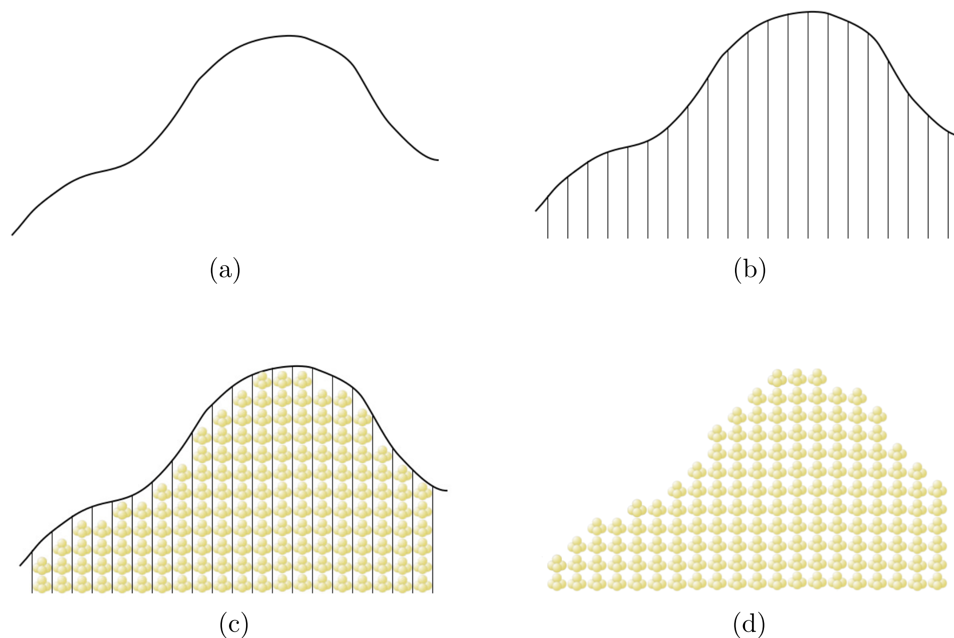


Figure 5.4: A cross section of terrain (a) is subdivided (b). Particles are inserted at each point of the subdivided terrain(c), resulting in a volume of sand which approximates the outline of the terrain (d).

particles are inserted into the arrays, until the height of the terrain at that point in the grid is reached. Each particle inserted also has a momentum and the relative position to its rigid body. Each rigid body also has a linear momentum, angular momentum and orientation. Once this process is complete, each rigid body and particle position is assigned an index, leaving us with the required arrays of particle and rigid body attributes, which are ready to be inserted into the particle system. The topmost particle in each stack is assigned a random orientation, which helps create a more natural looking surface.

These attribute arrays are passed to an inactive particle simulation. The typical size of these particle systems is up to 300,000 particles for denser particle systems. Each corresponding entry in an attribute array consists of four 32-bit floating point values, to ensure correspondence with the texture format. There are three attribute arrays for the set of particles, and four attribute arrays for the set of rigid bodies. All this data cannot be inserted into the system at once, as the CPU – GPU bus would be bottlenecked, resulting in a noticeable stuttering effect. Instead, the data is uploaded to the attribute textures over multiple frames, which significantly reduces the stuttering.

Algorithm 10 Overview of the conversion from the heightfield-based representation to the particle-based system.

Create particle arrays
for Each point on the heightfield section **do**
  Calculate the height at that point
  Create particles up to that height
Insert particle arrays into particle system
Each frame, insert a new particle attribute
for 1 second **do**
  Run the particle system without showing it
Show the particle system (alpha blend it in)

The particle system is then left to settle. Once injected into the system, the rigid bodies shift from their initial positions, often producing a wave-like effect as the lower particles adjust to the pressure applied by the particles above them. This appears unnatural, as the particle system should be representing terrain at rest. Introducing a settling period effectively negates this issue. The particle system is left to settle for one second before it is displayed to the user. We obviously want there to be as little delay as possible between adding the particles to the particle system, and displaying them to the user. We found that one second gives the particles enough time to settle. Any shorter than this, and the particle system still appears to move when presented to the user.

Once the particle system has settled, the particle-based terrain manager alpha blends in the particle system over the course of one second. However, rendering order is an issue. If the terrain is rendered first, particles which lie beneath the level of the terrain will be discarded, yet they will be visible once the blending is complete, which will lead to a noticeable popping effect. Additionally, particles which intersect with the terrain will only be partially rendered, resulting in unsightly artefacts. If the particle system is rendered first, then the terrain will not be rendered behind the particles, and the particles will appear to pop in, and blend between black and their resulting colour. Our solution is shown in Algorithm 11. First, the terrain is rendered as a first pass. The depth buffer is then cleared, and the particle system is rendered. The particles will thus be correctly blended with the underlying terrain. However, particles which should not be visible will be rendered. To address this, the terrain is rendered again, thus occluding particles which should be occluded. Note that during the first rendering pass of the terrain, shadow

mapping is disabled. This is because the particles will be shadowed, and if shadowing is enabled in the initial terrain pass, the particles will effectively shadowed twice.

Algorithm 11 C++ code snippet showing the rendering order of the terrain components. Note that the second argument for the heightfield rendering defines whether the heightfield should be rendered with shadows.

```
//Skybox will be occluded by all other objects
m_skybox->Render( m_renderingContext );
m_terrainManager->RenderHeightfield( m_renderingContext, false );

glClear( GL_DEPTH_BUFFER_BIT );

m_terrainManager->RenderParticleSystems( m_renderingContext );
m_terrainManager->RenderHeightfield( m_renderingContext, true );

//Render the other objects in the scene
//...
```

5.4 Conversion from Particle System to Heightfield

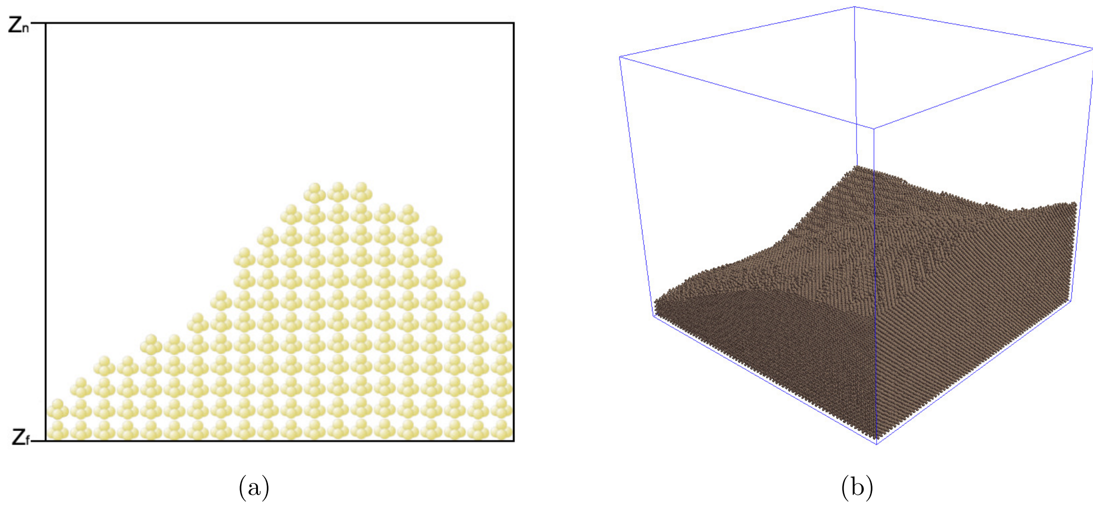
In order to convert the terrain from the particle-based representation to the heightfield-based representation, a method is required which is capable of quickly converting the volume of particles to a heightfield. A naive approach of sampling the particle system textures would be too costly, as it would require first transferring texture data from the GPU, and then iterating over each particle to find if it is the highest particle within a certain range in the x-z dimension. This would be computationally expensive, as it would require thousands of particles to be queried for each point on the terrain, and would also require an expensive texture transfer. An alternative method is thus required to maintain a high framerate and high fidelity.

Surface extraction from a set of points is a difficult problem, and many techniques have been developed to solve it, such as Gumhold et al.[18], and Rosenthal et al.[43]. However, these techniques typically take a few seconds, to minutes to complete, and thus are not suitable for real-time applications.

We note that a useful feature of the particle system is that while dynamic updates are expensive, the rendering of the system is comparatively cheap. Also, the particles form a single volume of sand. Based on these two observations, we have developed a novel rendering based technique to perform this conversion. By performing a top-down orthographic projection of the system, and extracting the depth buffer from the resulting image, a depth map of the terrain is obtained. As the position of the camera above the terrain is known, it is fairly simple to convert this resulting depth map into a heightmap. As the z-buffer sampling for an orthographic projection is linear[3], the following formula is used to convert the depth value to a height value:

$$height = y_{camera} - n - z(f - n)$$

where y_{camera} is the height of the camera, n is the distance to the near plane, f is the distance to the far plane and z is the depth value.



(a) (b)

Figure 5.5: (a)Shows how the camera is set up to perform the orthographic projection. Z_n is the near plane, and Z_f is the far plane. (b) Shows an actual particle system, with the view frustum for the orthographic projection drawn in blue.

The orthographic projection is centred on the middle of the particle system, and the clipping planes are extended to cover the entire particle system. A 32-bit depth texture is attached to the framebuffer, and colour attachments are disabled. The y-position of the camera is to 10 units above the maximum height of the terrain. In this way we are

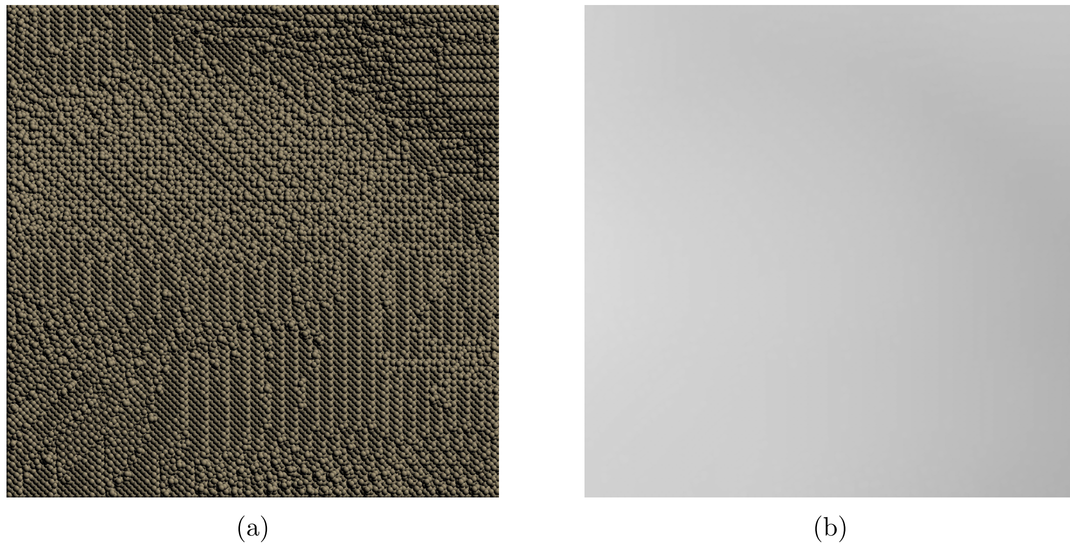
assured that the rendered image will contain the entire particle system. Figure 5.5 shows an example of how the camera is set up.

The particle system adjusts the particle sizes to correct for parallax scaling (i.e. objects closer to the camera appear larger than objects further away from the camera). Unfortunately, this adjustment results in particles closer to the camera being far larger than particles further from the camera when rendered with an orthographic projection. In order to avoid this, the particle system is rendered with fixed particle sizes during this pass.

Algorithm 12 Overview of how the conversion from particle system to heightfield is processed.

```
Perform top down orthographic projection
Extract depth texture
for Each point on the heightfield do
    Calculate height at that point from the depth texture
    Inject the height into the heightfield
Apply Gaussian filter to converted area
Reset clipmaps to propagate changes
```

Performing this rendering step with a resolution equal to the size of the particle system produced poor results. Instead, the system renders the particle system using a resolution of 1024×1024. See Figure 5.6 for an example of the resulting images obtained from the top down orthographic projection. The resolution used is much higher than the number of visible particles, regardless of the particle system scale, with each particle covering multiple pixels. In order to compensate for different particle scales or different particle system bounds, the sizes of the particles are adjusted. The depth map is then sampled at points which correspond to points in the underlying heightmap. This produces far better results. However, as can be seen in Figure 5.7, a stepping effect may occur, since, depending on the positions of the particles, the same particle may be sampled multiple times. While the overall profile of the heightmap appears correct, the resulting terrain is noticeably tiered. This is corrected by applying a Gaussian filter[46], once the heightmap section has been inserted into the terrain, in order to smooth out sharp transitions. We found that a simple filter with a 3×3 kernel sufficed. Using a filter with a larger kernel size resulted in over-smoothing, thus losing the detail of changes made to the terrain in the particle system.

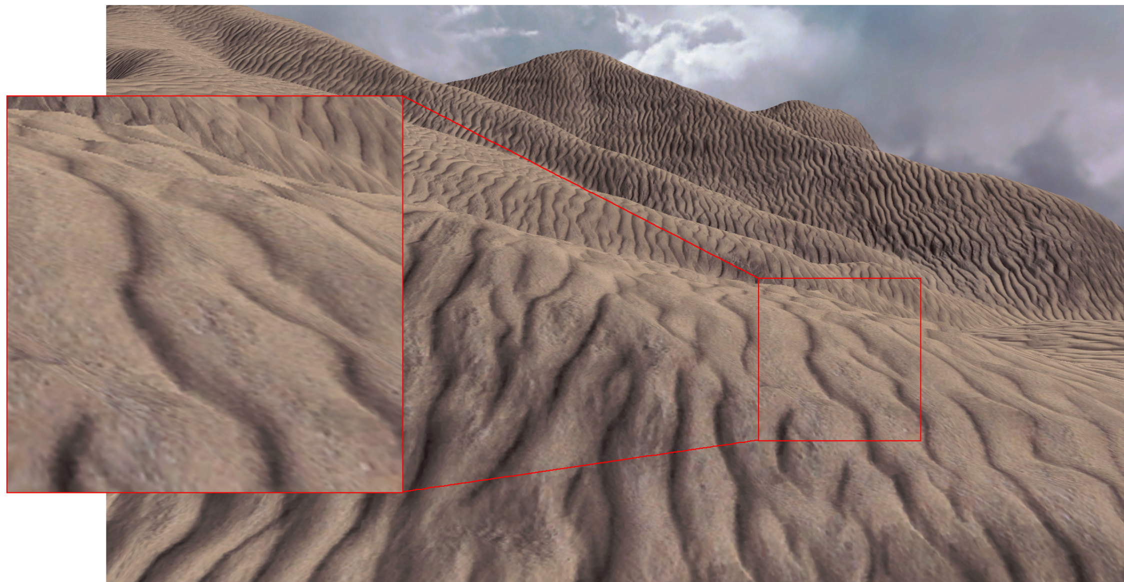


(a) (b)

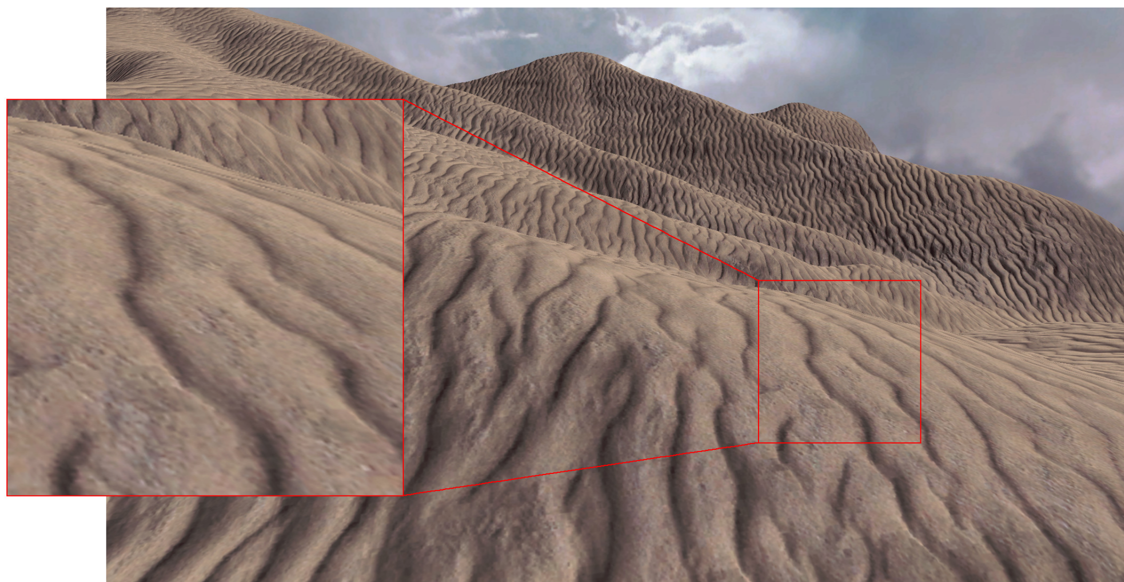
Figure 5.6: The result of the top-down orthographic projection used to convert between the particle-based and heightfield-based representations. (a) Shows the colour map and (b) shows the depth map.

The resultant heightmap is inserted into the terrains heightfield, over-writing the section of the heightfield which corresponds to the particle system. This causes any changes made to the terrain in the particle-based system to persist in the heightfield-based system. However, as can be seen in Figure 5.7a, there may be a discontinuity along the edges of the newly injected heightmap. This is due to a sudden transition from the original heightfield, to the heightfield representing the particle system, which may lie slightly below or above the level of the terrain at that point. In order to correct this, a Gaussian filter with a 3×3 kernel is applied along the border of the inserted heightfield. This filter is also applied across the entire injected heightmap. As shown in Figure 5.7b, this effectively eliminates the stepping artefacts in the injected heightmap, and smooths the transition between the pre-existing terrain, and the newly injected region.

Finally, once the converted section of terrain is added to the terrain, the terrain clipmaps are reset. This means that each texture clipmap is overwritten with data from the newly updated base terrain. This entails rendering a single texture mapped quad per clipmap level, and resetting the texture origin, and is thus not computationally expensive. This propagates the changes to the terrain through the various clipmap levels, and means that changes made to terrain in the particle system persist in the heightfield-based terrain.



(a)



(b)

Figure 5.7: An example terrain with a converted particle system inserted. The inset shows a zoomed in view of the same area in both figures. (a) Shows the result of the injected heightfield, with no filtering applied. This results in an obviously bumpy surface. (b) Shows the same section, but with Gaussian filtering applied. No discernible discontinuities are present.

5.5 Particle-Based Representation for Models

In order for dynamic objects to interact with the particle-based terrain, they must have an associated particle-based representation. We thus convert each model in the scene to a particle-based representation. This is achieved by creating a signed distance field[47] for each model. Computing a signed distance field is expensive[36], so we perform this conversion as a pre-processing step.

A signed distance field represents a model as a grid based sampling, with each point in the grid representing the distance to the closest geometric primitive in the model. Points inside the model are assigned a negative value, whereas points outside are assigned a positive value. In order to obtain a signed distance field, a model is overlayed with a regular grid. For our application, the unit size of the grid is proportional to the size of the particles within a specific particle system. This is required, as the signed distance field will be used to generate a particle-based representation of the object, and these particles need to be correctly spaced in order to interact with the granules in the particle system. The grid extends two units beyond the minimum and maximum bound of the object in each dimension. This creates a buffer region around the object, so that the border of the object may be calculated at a later stage. The distance to the nearest surface of the object is then computed for each point on the grid. This distance is negated if the point lies within the object. By comparing the sign of the distance value with those of adjacent nodes in the grid, the system is capable of quickly identifying points on the grid which lie on the border of the object.

Computing the signed distance field is a simple, yet computationally expensive task. For each node in the grid, the distance to each triangle in the model is computed. To find the distance to a triangle, first the point is projected onto the triangles plane. If the result lies within the bounds of the triangle, then this is the shortest distance to the triangle. This is determined by calculating the barycentric coordinates[27] of the intersection point, with regards to the triangles vertices. If the point lies outside the triangle, then the distance to each edge of the triangle is computed, as well as the distance to each vertex of the triangle. The smallest of these values then represents the shortest distance to the triangle.

Once the distance value has been calculated, the sign needs to be computed. There are multiple approaches to this, as shown by Jones et al.[25]. We use the simple approach of

casting rays along the z-axis. Grid points, where an odd number of intersections with the mesh have occurred, are inside the mesh, and are thus assigned a negative sign. All other points are assigned a positive sign.

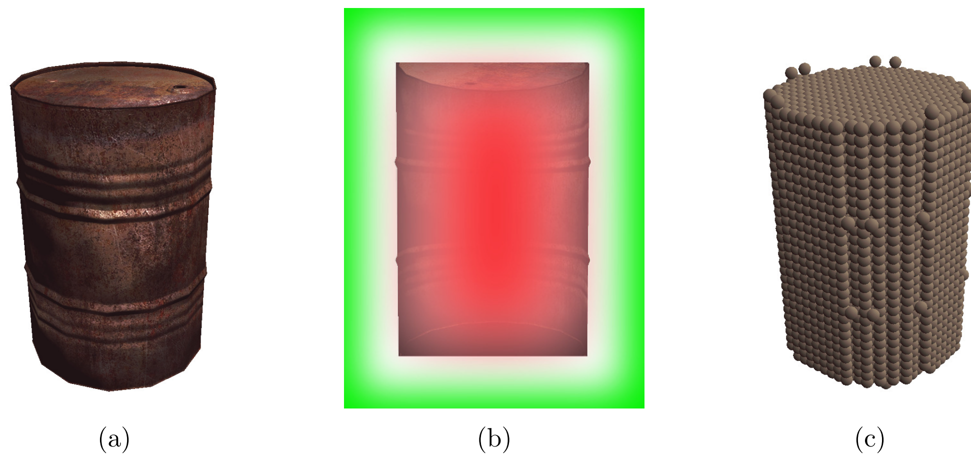
Note that far more efficient techniques for computing signed distance fields exist, such as that of Erleben and Dohlmann[15]. However, the efficiency of this transformation is not of primary concern to us, and thus we have implemented the simpler method that we have presented here.

Once the signed distance field has been generated, particles are created at any negative value in the distance field, which lies adjacent to a positive value (i.e. at each point on the border of the object). This creates a particle-based representation of the object. The particles together form a single rigid body within the system, in the same way that multiple particles constitute a granule.

This creates a surface layer of particles which represent the object. This is sufficient, as even if we added particles throughout the body of the model, collisions with external particles would first occur with these surface particles, therefore preventing the internal particles from colliding with external particles. In fact, by using a single outer layer of particles to represent the objects, both memory and computational resources are saved.

Whilst the particle-based representation should interact with the terrain, the model should still be rendered in its original form. The mass of the object is calculated from the number of constituent particles. Obviously, various objects have different densities, so this may result in an incorrect representation. The objects may thus also be assigned a specified weight. This particle-based representation is then stored by the particle-based terrain manager, along with the model information, such as the weight and the number of constituent particles, so that particle-based versions of objects may be injected into the particle systems as required.

The LOD system uses particle systems of different scales. A separate particle-based representation of an object needs to be computed for each scale within the simulation, as the different particle densities require different representations in order to interact correctly with the terrain.



(a) (b) (c)

Figure 5.8: Conversion of a model to a particle-based representation. (a) Shows the model. (b) Shows a cross section of the model, overlayed with the signed distance field. The red colour indicates negative values, and green colours indicate positive values. The intensity of the colours indicates the magnitude of the value.

As converting a model to the particle-based representation is computationally expensive, it is stored in a file once it has been processed. Then, whenever a model is loaded from file, the system looks for the corresponding particle-based representation file in the directory. If it finds the file, it simply loads the particle-based representation of the model from this file instead of recalculating it.

Unlike terrain, particle-based models require a relatively small amount data to be uploaded to the GPU. Thus, the data is inserted all at once. Also, as the model only consists of one rigid body, no settling period is required. However, before the data is inserted into the particle-system, it first needs to be rotated and translated, in order to match the position and rotation of the dynamic object. Additionally, the momentum of each particle and the rigid body needs to be set to match that of the dynamic object.

5.6 Summary

This chapter examined the terrain manager, which forms the core of the level of detail system. This component manages the various particle systems, determines which are active, and it is even capable of disabling updates to the particle system, once a rigid-body has

come to rest. This creates the illusion that there are more active particle systems than there actually are. These particle systems can the be re-enabled once another dynamic object is added. Denser particle simulations are used closer to the observer, whereas coarser simulations are used further away, as less detail is required further away from the camera.

The terrain manager also handles the conversion of the heightfield-based terrain to the particle-based representation, and vice versa. These conversions happen in real-time. When converting from the heightfield-based terrain to the particle-based representation, a slight delay is added, to prevent the scene from stuttering due to all the data being transferred to the GPU, and to allow the particle system to settle before being shown. The particle-system to heightfield conversion is achieved by using a novel top-down orthographic rendering technique, whereby the camera is placed above the particle system, and the particle system is drawn to an off screen buffer. The depth buffer is then sampled, and it is then fairly arbitrary to convert this value to height value, which can then be inserted into the terrain heightfield. Additionally, a Gaussian filter is applied to the result, which smooths out any discontinuities which may arise.

Finally the terrain manager also manages the conversion of models to a particle-based representation. This allows the models to interact with the particle systems. This is achieved by creating a signed distance field representation of the object. Particles may then be inserted at any negative point which is adjacent to a positive value in the distance field. This technique is computationally expensive, but only needs to be run at start up. The result may be stored in a file, so that it only ever needs to be run once for each particle scale.

Chapter 6

Evaluation and Results

The system we have developed is capable of simulating large-scale granular terrains in real-time. Additionally, it is capable of switching between a heightfield-based representation and a particular simulation in real-time, whilst ensuring that changes to the terrain persist between representations. This chapter presents results to substantiate these claims.

In the interests of clarity, the results are given for the individual system components, as well as the final integrated system. First, results for the GPU Geometry implementation are provided. Then the particle-based simulation framework is analysed. Finally, the integrated LOD framework is analysed.

The principle focus of this thesis is a proof of concept, to show that such a system may be feasible for use in games and simulations. Therefore our primary concern is performance results. Memory usage is also a very important, as it effects how many particle systems may be active at any time, and effects the system's potential usefulness for games and simulations. Additionally, three scenarios have been developed in order to test the various facets of the LOD framework.

When evaluating performance results, it is important to understand the terms framerate and real-time, which are closely related. Framerate is a performance measure, and is expressed as the number of frames rendered per second (fps). Real-time refers to the ability of a system to respond to changes within a specified time frame. As the time between frames is effectively indicated by the framerate, this can be expressed as a framerate. As noted by Hasenfratz et al.[21], there exists no strict definition for real-time framerates, although they consider 10 FPS the lower bound. We prefer to aim for 30 FPS, as this has been shown to increase the feeling of presence in virtual environments[35].

6.1 Testing Environment

The system was implemented in C++, using the Microsoft Visual Studio 2010 IDE. The system was run on Windows 7 Service Pack 1 and employed OpenGL 2.1, with GLEW 1.10.0 to provide access to advanced GPU features which are not natively accessible using OpenGL 2.1. SDL 1.2.15 was used to manage windowing, the creation of the OpenGL context and user input. The nVidia Cg shader library was used to implement the shaders for the GPU geometry clipmaps implementation, whereas GLSL was used to implement the shaders for the particle simulation framework. All results were generated at a resolution of 1920x1080.

The hardware testing platform consisted of an Intel Core i7 930 processor with 6GB of DDR3 memory in a triple channel configuration. Results are presented for both an nVidia Geforce GTX 460 with 1024 MB RAM, and an nVidia Geforce GTX 770 with 2048 MB RAM. These two graphics cards where chosen to represent the performance of modern mid-range and high-end cards respectively. Table 6.1 shows a comparison of the specifications of the two cards. The nVidia display driver 331.58 was used for both cards.

	Geforce GTX 460	Geforce GTX 770
Streaming Multiprocessors	7	8
Shader Cores	336	1536
Texture Units	56	128
ROP Units	32	32
Core Clock	675 MHz	1046 MHz
Shader Clock	1350 MHz	1046 MHz
Memory	1024MB GDDR5	2048 MB GDDR5
Memory Clock	900 MHz	1750 MHz
Memory Data Rate	3600 MHz	7000 MHz
Memory Interface	256-bit	256-bit
Memory Bandwidth	115 GB/s	224 GB/s

Table 6.1: Comparison between the nVidia Geforce GTX 460 and the nVidia Geforce GTX 770. The GTX 770 is a more powerful, more modern card, and as can be seen from the table, has far more computational resources.

6.2 GPU Geometry Clipmaps

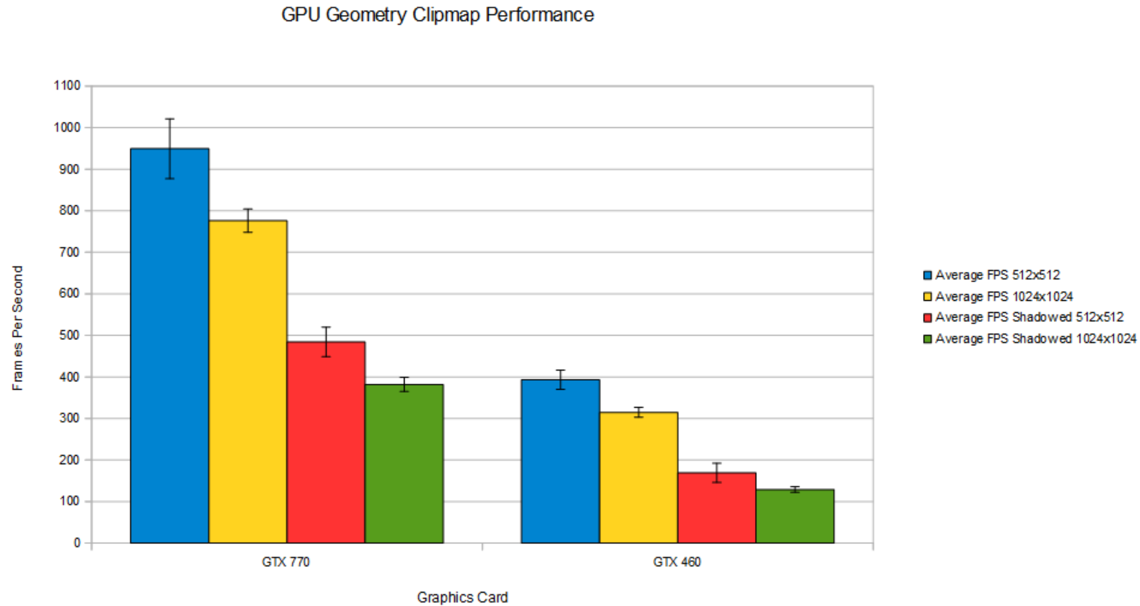


Figure 6.1: GPU geometry clipmaps performance results for the nVidia GTX 770 and nVidia GTX 460, both with and without shadows enabled. The standard deviation is shown by the black error bars.

In order to evaluate the performance of the GPU Geometry clipmaps implementation, we tested the performance across a range of different terrains. Ten different terrains are used in total, and the results are averaged, to provide a reasonable measure to evaluate system performance. Each terrain was run five times. Two different terrain sizes are used: 512×512 and 1024×1024. The view of the terrain is from the corner, and spans the entire terrain. The 512×512 terrains produce 79,886 triangles, whereas the 1024×1024 terrains produce 103,700 triangles. The size of the terrain has a fairly minimal impact on performance, as only a single additional clipmap level is required to render the extra terrain section in the larger terrain, due to the exponential increase in the grid resolution.

As seen in Figure 6.1, the systems performance is exceptionally good, both with the midrange nVidia GTX 460 and the high-end nVidia GTX 770. Such high performance from the heightfield-based component of the system is necessary, since the particle-based simulation is extremely computationally expensive (see Section 6.3).

The transition region between the various clipmap levels is also important: there should be no apparent visible seams, and the user should not be aware of the LOD scheme. As Figure 6.2 shows, the terrain exhibits no visible seams across clipmap borders. However, when moving quickly across terrain that contains jagged features, the LOD scheme is apparent to the user. This is because finer levels are more detailed than coarser levels. Thus, as an area of terrain moves into the finer clipmap levels, these finer details become apparent to the user. For instance, a jagged peak may fall between vertices in the coarser level, thus effectively being smoothed over, but this peak will be apparent within the finer levels. As it moves through the clipmap blending region, it will appear to geomorph in, which may be observed by the user. However, as the primary focus of this thesis is on simulating sandy terrains, which lack such sharp contours, this is deemed to be acceptable.

In order to evaluate the visual accuracy of the terrains, we rendered terrains with and without geometry clipmaps enabled, so that the results can be compared. In the interest of readability, these images are presented in Appendix B.

Figure 6.2: A transition between two clipmap levels. There are no noticeable visual seams present.

Finally, the memory usage of the system is measured. The system uses a total of 30MB of main memory and 40MB of graphics memory. A low memory footprint is important, as it allows for other high quality assets to be used in the game or simulation.

6.3 Particle-based Simulation

For this work, the performance of the particle-based terrain simulation is of greatest importance. The physical correctness of the system has already been established[29].

To test the particle-based system, particles are injected into a 3D cuboid. Particles are injected in sets of 40,000, i.e. 10,000 granules or rigid bodies at a time. The framerate is measured for each set of injected particles.

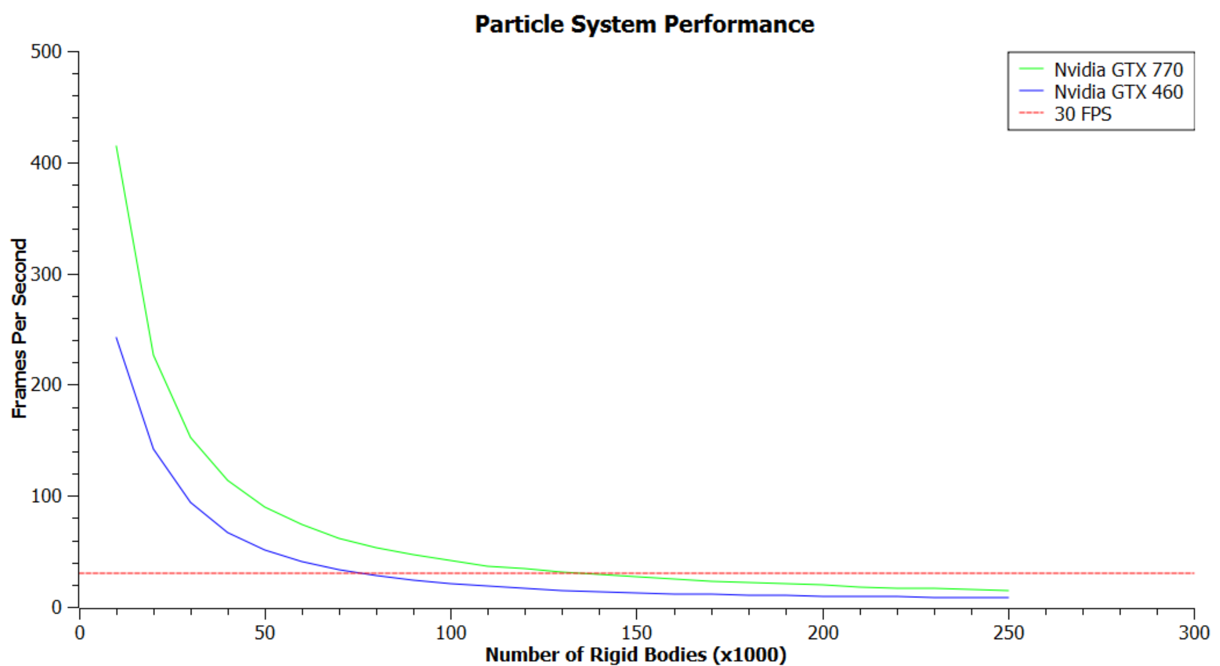


Figure 6.3: Performance results of the particle-based simulation for both the nVidia GTX 460 and the nVidia GTX 770. Note that each rigid body, or granule, is made up of four particles in a tetrahedral configuration. The 30 FPS line is denoted in red, indicating how many granules may be rendered in real-time.

The system scales fairly well with the computational power of the graphics card used, with the more powerful GTX 770 achieving almost double the performance of the GTX 460. The framerate is inversely proportional to the number of particles in the system.

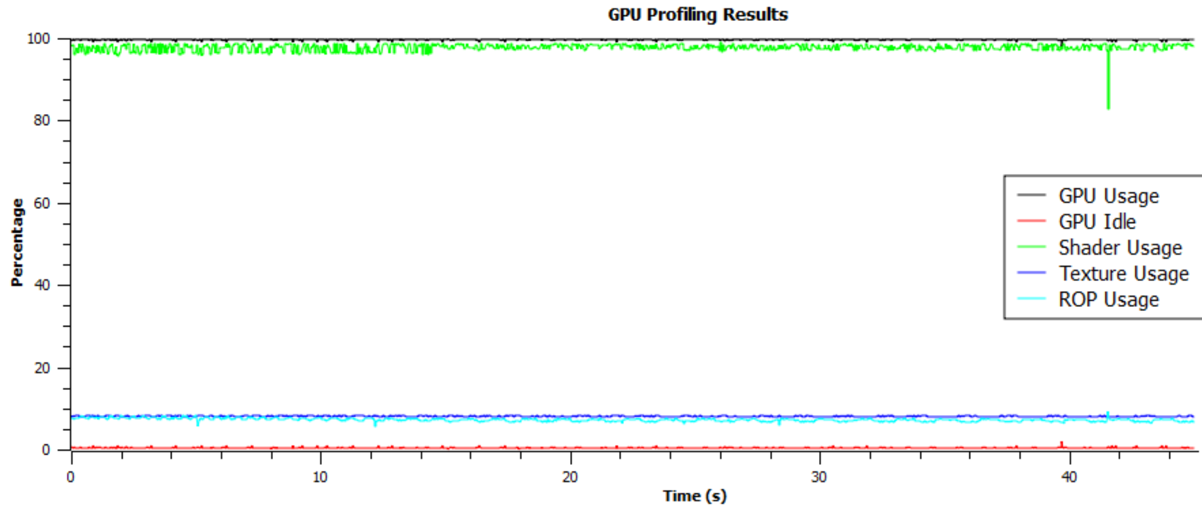


Figure 6.4: The performance counters generated using nVidia Perfkit 3.1.0.13233 for a particle system with 100,000 granules (400,000 particles), over a 45 second period. As we can see, shader usage is very high, and there is very little GPU Idle time. Comparing this to the other performance counters, it is obvious that shader performance is the bottleneck to system performance.

.

Using nVidia Perfkit 3.1.0.13233 to measure the GPU performance counters, it is apparent that shader usage is extremely high, and is the primary bottleneck behind GPU performance. The next closest performance metric, texture usage, peaks at less than 10%, which means that system performance should continue to scale well with increases in GPU shader performance.

Given the framerate, the time to render a frame (frame time) can be calculated, as $t_{frame} = \frac{1}{frames\ per\ second}$. In order to analyse the performance, two performance counters are measured: the total system performance and the rendering performance. Rendering performance is measured by disabling the particle system updates. From these two measures we can infer the update performance, as $t_{frame} = t_{update} + t_{render}$. The frame time, seperated into these two components, is shown in Figure 6.5 for the nVidia GTX 770.

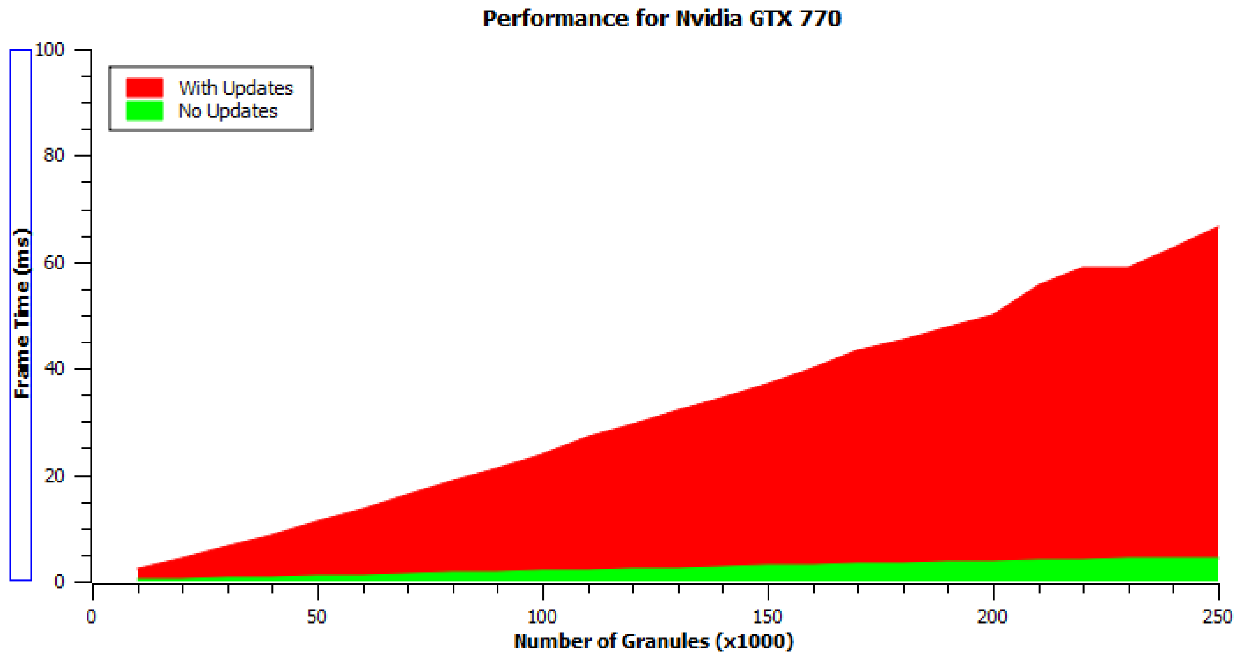


Figure 6.5: Frame time of the particle-based simulations, with updates enabled and disabled, in order to show the computational resources that are used to render the particle system, and process the updates. The time spent to process updates is indicated by the red area, whereas the time spent to render the particle system is shown in green.

Clearly, the update performance is the major limiting factor behind system performance. This is to be expected, as the updates entail mapping the particles to a 3D grid, then performing collisions for hundreds of thousands of particles, and updating the rigid bodies and particles in response to these collisions. The frame time, both with and without updates, is linearly proportional to the number of particles, which accords with the previous performance results.

The performance results from Longmore[29] are shown in Figure 6.6. Comparing the results to those in Figure 6.5, an increase in system performance can be seen with our results. This is mainly due to the graphics card used, as the nVidia Geforce 8800 GTX used by Longmore is far less powerful than the nVidia Geforce GTX 770 that we have used.

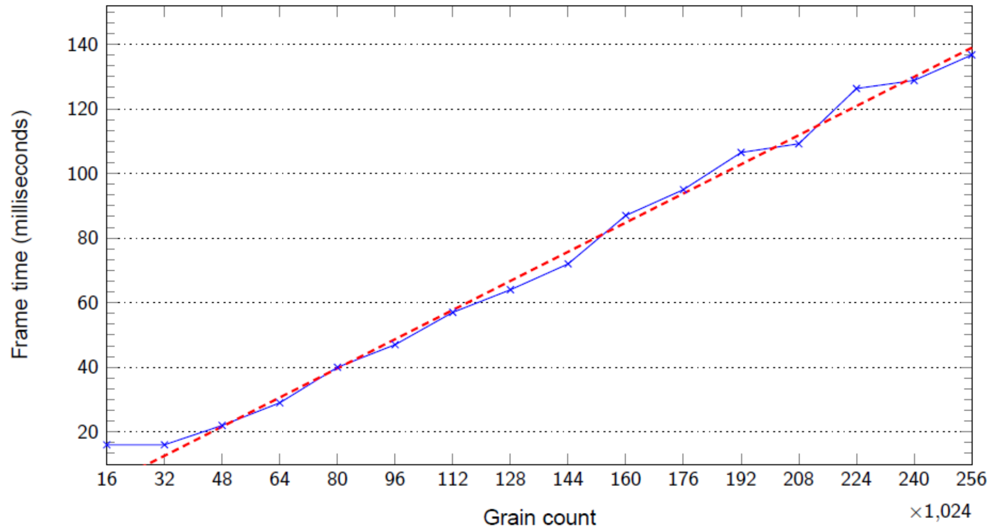


Figure 6.6: Performance results from Longmore [29]. The red dashed line represents a linear fit of the data. The results were generated using an nVidia Geforce 8800 GTX. Comparing these results to the results from Figure 6.5, we note that 100,000 particles results in a frame time of 25ms with the GTX 770 and 50ms with the 8800 GTX. Thus, the particle system has scaled well with the increase in computational performance.

Shadow mapping performance is shown in Figure 6.7. It is readily apparent that the framerates are much lower than with the regular rendering set up. This is because each fragment must be mapped to shadow space; the basic method computes the transformation at each vertex and interpolates the result between vertices. This is because each particle is rendered as a separate point, and thus there are no additional vertices with which to interpolate the result. Shadow mapping for such a particle is therefore extremely computationally expensive, although the GTX 770 maintains real-time frame rates up to approximately 80,000 rigid bodies (i.e. 320,000 particles).

The system used 140 MB of main memory and 188 MB of GPU memory. This memory usage is mainly due to the number of textures required to store all the particle and rigid body attributes. Fortunately, with the amount of memory available in most modern computers and graphics cards, this memory usage is perfectly acceptable, and allows us to use multiple particle-simulations concurrently, along with other graphical assets. Note, that the number of particles added to the system does not affect the memory usage. Textures of a set size are used to hold the particle and granule attributes. These textures are allocated during the system set up. Thus the memory usage is independent of the particle count.

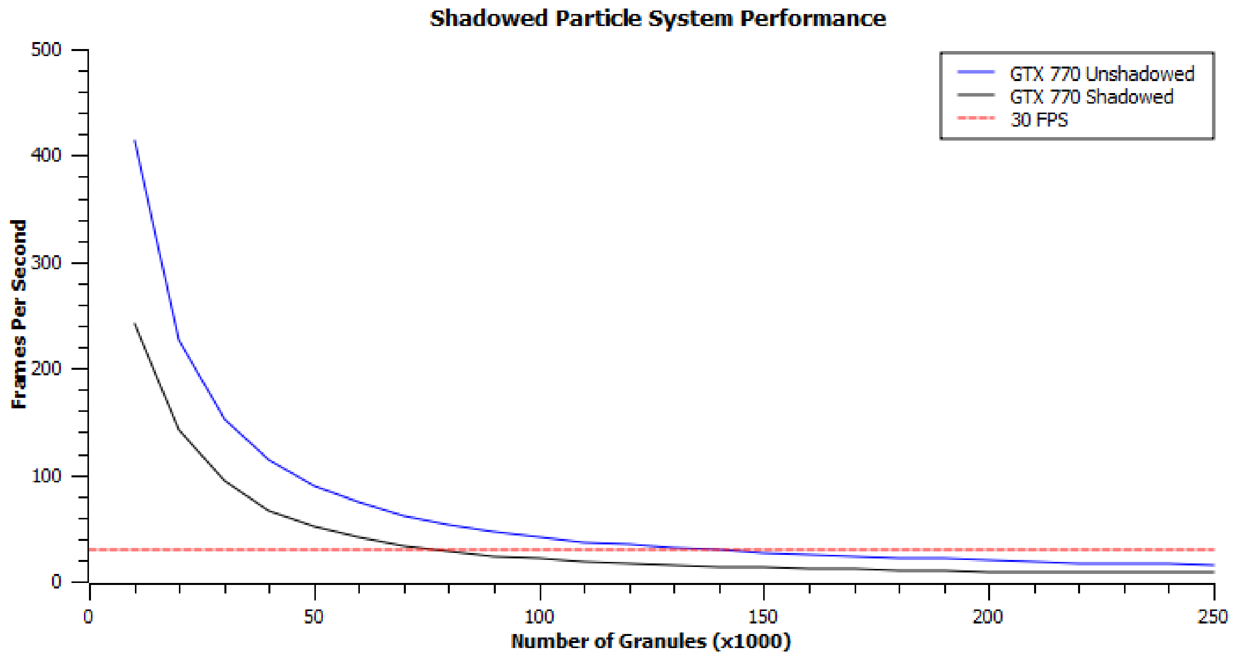


Figure 6.7: Performance results of the particle-based simulation with shadow mapping enabled, for the nVidia GTX 770. The red line shows the 30 FPS framerate, which indicates real-time performance. Comparing these results to those in Figure 6.3, we see a reduction in the rendering performance. However, the system is still able to simulate approximately 80,000 granules in real-time.

6.4 Integrated System

The particle-based simulation framework is designed to be used in a wide variety of circumstances. Thus, evaluating system performance is difficult. In order to test the integrated system performance, three test scenes were developed, with each scenario designed to test a different facet of the system. Note that we have made videos of the test scenarios available online (see Appendix A). Each test scenario was run five times, with the results averaged to obtain the final result.

6.4.1 Test Scene 1

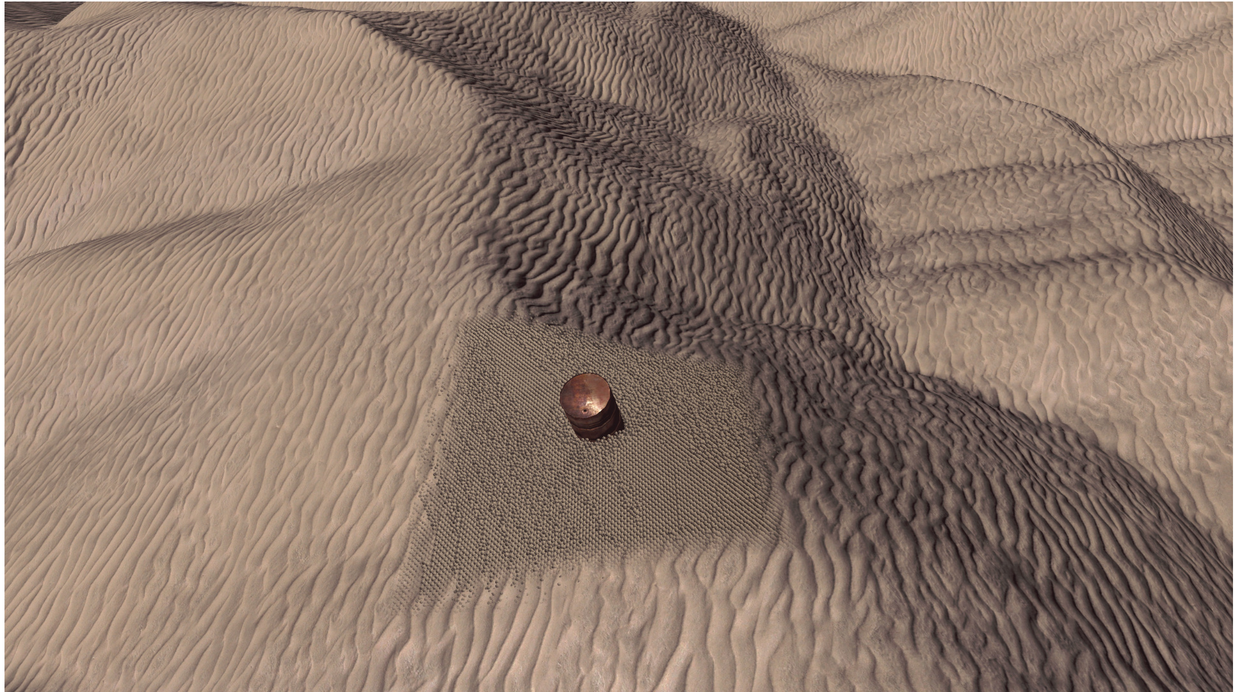


Figure 6.8: Screen shot of the first test scene. The particle simulation can be clearly seen interacting with a model. This system contains a particularly high density particle system, in order to stress test the system performance.

The first scenario contains one extremely high quality particle system. This is a stress test, to test the systems ability to render and simulate a large particle system, while also rendering the heightfield-based terrain, all the while maintaining real-time performance. The particle system for this scenario consists of 115,502 granules (i.e. 462,008 particles).

In addition, a barrel model is dropped onto the particle simulation. This model is made up of 6,261 particles. This is done to showcase the realistic interactions, and puts further stress on the system. The performance results for this scenario are shown in Figure 6.9.

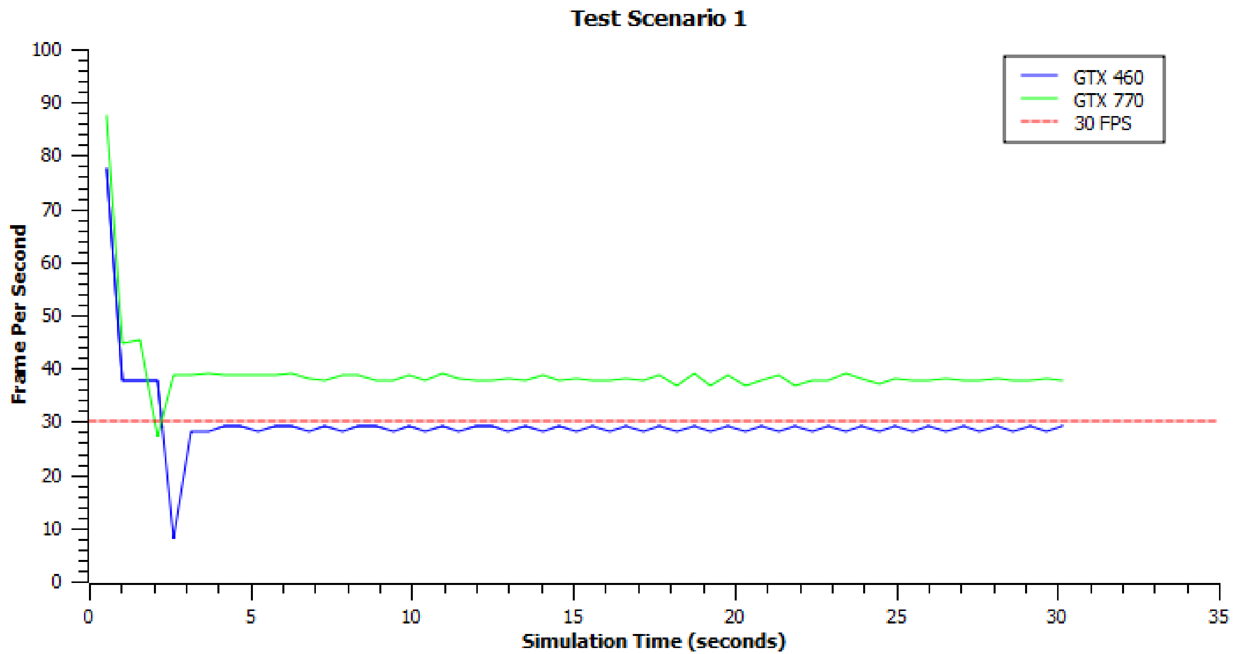


Figure 6.9: Performance results for the first test scenario, for both the nVidia GTX 460 and the nVidia GTX 770. The GTX 460 hovers around 29 FPS, which is just below real-time performance. However, this is a midrange card, and thus the result is acceptable. The GTX 770 easily delivers more than 30 FPS.

The nVidia GTX 770 maintains good performance, only slipping below the 30 FPS level during injection of the model. The nVidia GTX 460, however, hovers around 29 FPS. While not ideal, this is certainly still an acceptable result, given the number of granules present in the particle system. The size of a typical foreground simulation is 250,000 particles, midrange simulations typically contain 80,000 particles, and a simulation in the distance typically has 20,000 particles.

Comparing the results from this test scenario to the results from Figure 6.3, we note that the performance in this scenario is slightly better. This is because when the particles are injected into the system with our conversion method, the particles interlock in such a way that the number of collisions is reduced. This doesn't happen when the particles are simply dropped into a volume, as is the case in Section 6.3.

6.4.2 Test Scene 2

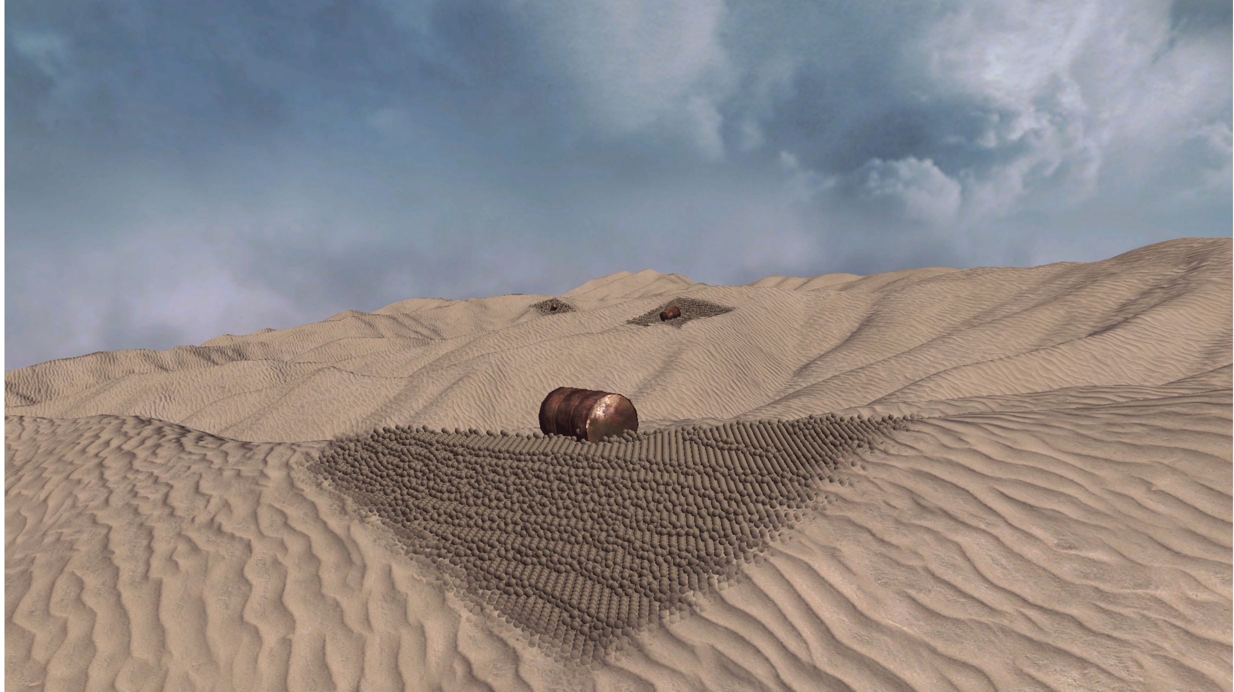


Figure 6.10: Screen shot of the second test scene. Three simulations can be seen, each with a different scale.

The second scenario contains three particle systems, with three different particle scales. Each scale represents a different distance from the camera. The point of this test is to verify that multiple particle simulations, of different scales, may be used concurrently, whilst maintaining real-time performance. This scenario represents the general envisaged use case, with a few simulations taking place at different distances from the user.

Figure 6.10 shows a screen shot from the test scenario. Each particle system in the scenario has a scale appropriate for the distance from the camera. The foreground simulation contains 202,352 particles, the midrange particle simulation contains 54,876 particles, and the far particle simulation contains 25,776 particles. The performance results for this scenario are shown in Figure 6.11.

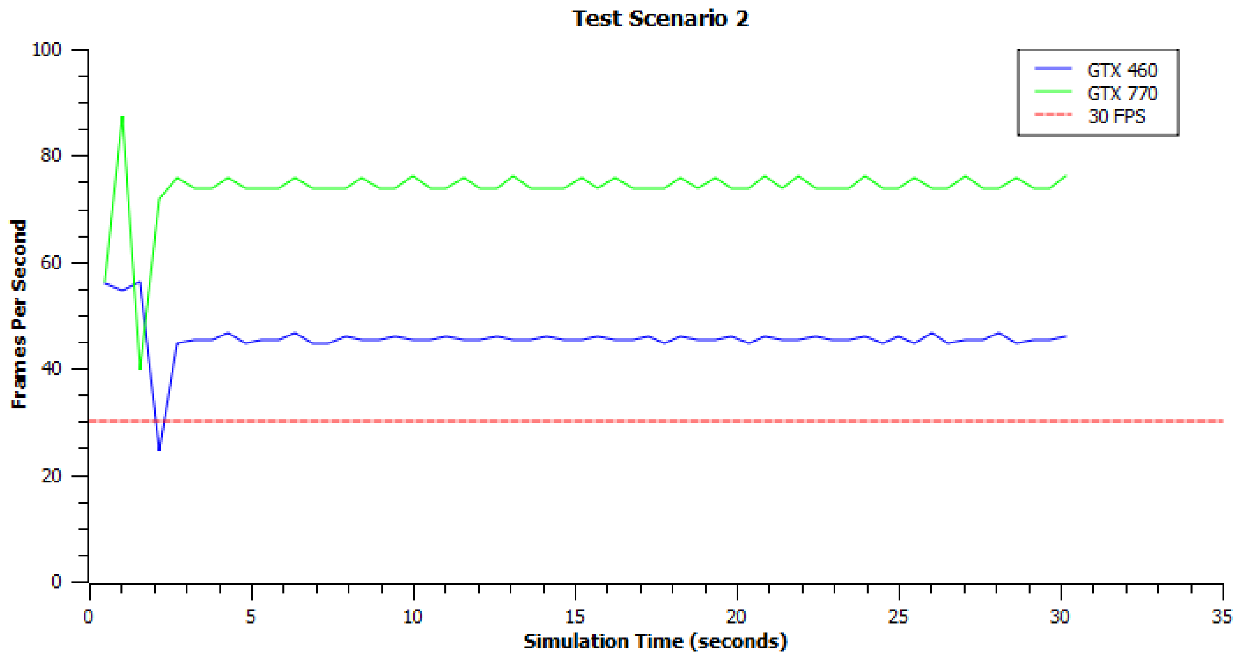


Figure 6.11: Performance results for the second test scenario, for both the nVidia GTX 460 and the nVidia GTX 770. Both cards show good performance, with the GTX 460 hovering around 45 FPS, and the GTX 770 hovering around 75 FPS.

In this scenario, both cards are able to maintain good framerate, with the GTX 460 only dipping below the 30 FPS level briefly, during model injection. The system has problems with spikes in frame rates during model injection. This transient effect is the result of all the particle data for the model being uploaded to the GPU in a single frame. Unfortunately, there is no way to combat this, as the particle system is active when the model is inserted, and thus the data cannot be inserted over multiple frames.

6.4.3 Test Scene 3

The final test consists of ten particle simulations. The dynamic object for each particle system is added ten seconds after the last. This shows off the ability of the system to handle many particle simulations, by disabling updates once bodies have come to rest. The number of granules in each particle system is shown in Table 6.2. A particle scale of 0.33 is used for each simulation. Figure 6.12 shows a screen shot of the scenario, at the end of the test.

Particle System	Granule Count
1	41292
2	42195
3	62773
4	63008
5	54339
6	67282
7	41964
8	46609
9	48128
10	60568

Table 6.2: Table showing how many granules constitute each of the particle-based simulations in the third test scenario.

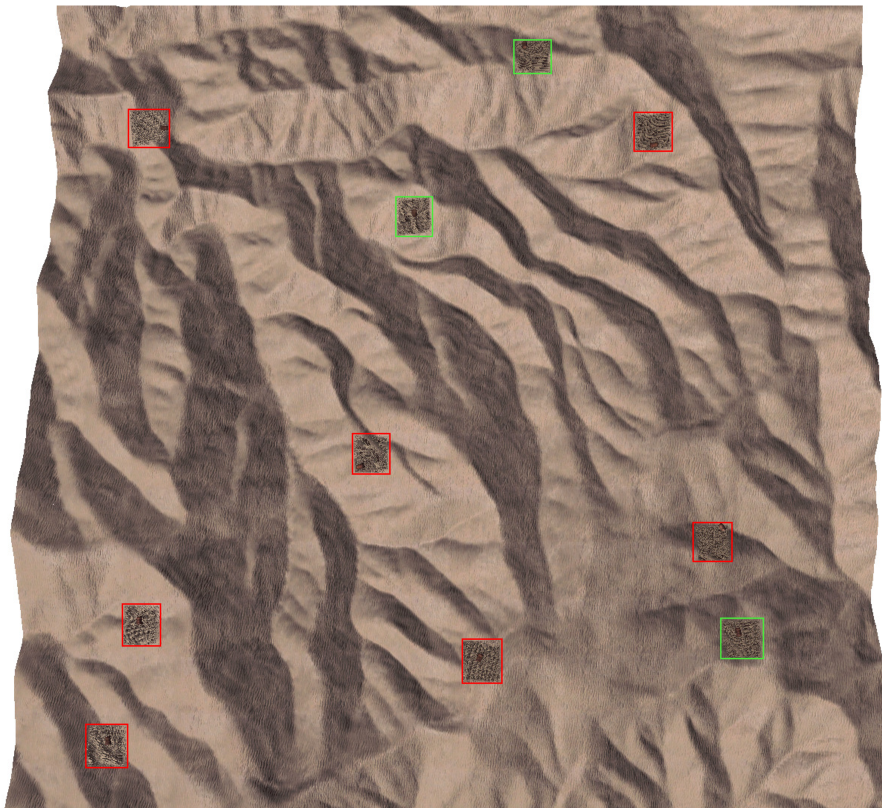


Figure 6.12: Screen shot of the third test scene. Ten particle simulations can be seen. The active systems are outlined in green, and the systems at rest are outlined in red.

The results are shown in Figure 6.13. Here the results are less promising. The GTX 460 in particular struggles with this test. Closer investigation showed that due to the slower updates the particle system models took a longer time to come to rest. By this time another particle system had been populated, thus competing for computational resources. This in turn caused the updates for both simulations to slow down, and when another simulation was added, the effect compounded, leading to a cascading drop in performance. The GTX 770 on the other hand, is powerful enough to ensure that this doesn't occur. For most of the test three of these simulations are active at the same time, which the GTX 770 is able to handle without any problems, unlike the GTX 460.

This has great implications for the use of the system in games and simulations for mid-range and lower level graphics cards. The particle systems used should not be too finely scaled, in order to allow the dynamic objects to come to rest quickly. Additionally, the number of concurrently active particle simulations may need to be limited.

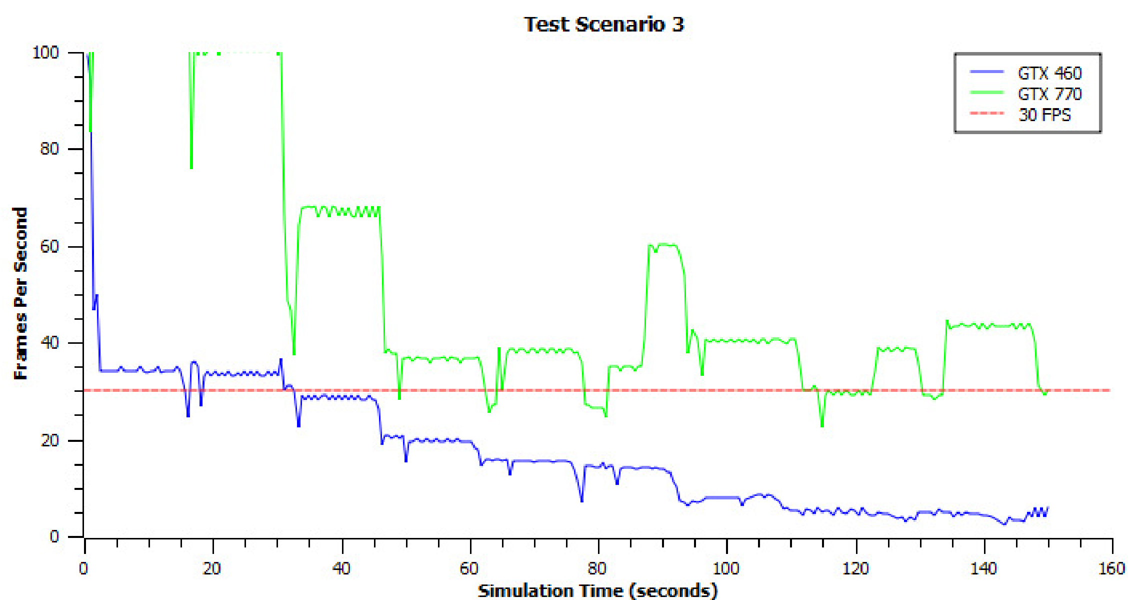


Figure 6.13: Performance results for the third test scenario, for both the nVidia GTX 460 and the nVidia GTX 770. The GTX 770 is generally able to handle the load applied by this test, only dropping below the 30 FPS level a few times, where as the GTX 460 is not able to maintain real-time performance. Note that between 1.5 seconds and 16.5 seconds, the GTX 770 attains more than 100 FPS. Likewise, between 18 seconds and 30 seconds, the performance hovers around 100 FPS. This is because only one particle system is active during these periods. We have limited the graph to 100 FPS, to maintain consistency with the rest of our results.

6.4.4 Further Tests

To verify the validity of the conversion between terrain representations, multiple areas of terrain are converted from the heightfield-based representation to the particle-based representation and back again. The difference between the new heights and the original heights are then calculated. These differences are averaged, in order to provide a useful measure for the error present in the resulting new terrain section. The test is also performed at multiple particle scales, in order to determine the effect that particle size has on the error in the resulting terrain.

As Figure 6.14 shows, there is a minimal difference between the original terrain heights, and the updated values, even at larger particle sizes, with a maximum average error of 0.411 meters occurring at the base particle scale. This ensures that the terrain may be freely converted between the two representations, without affecting the validity of the underlying terrain. It is also interesting to note that there is a linear relationship between the size of the particles used in the particle simulation, and the resulting error in the converted terrain section. Even at the largest particle scale, however, the error exhibited is acceptable, as the error value remains very small.

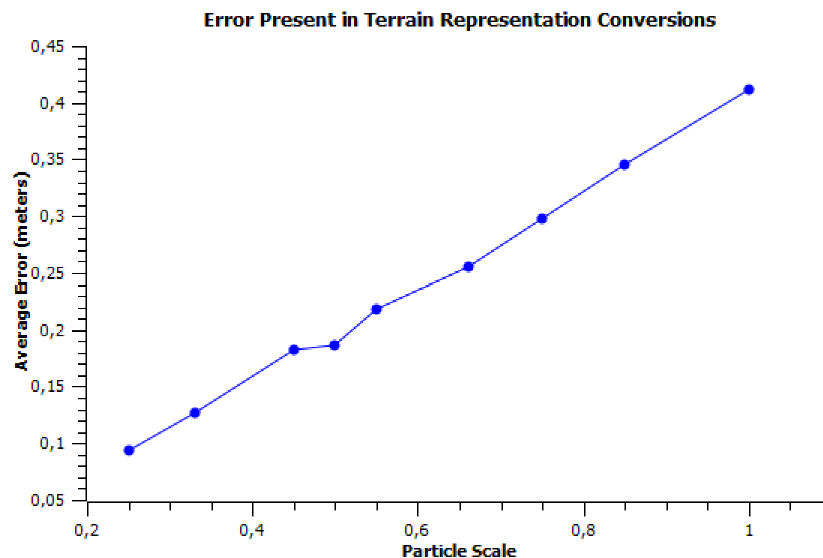
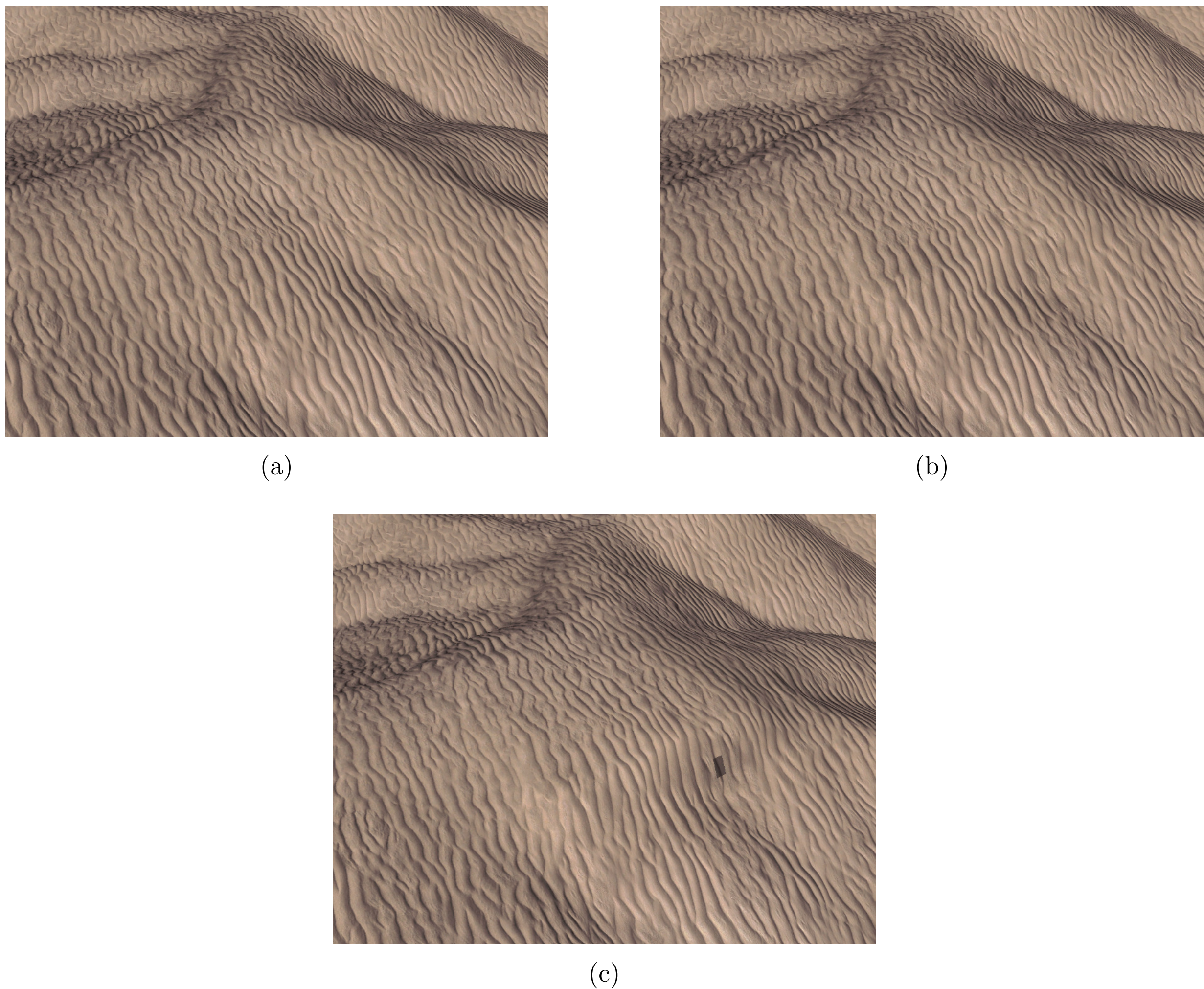


Figure 6.14: The average error present during terrain conversions at different particle scales. As we can see, minimal error is present, even at the coarser particle system scales.

Unfortunately, although slight, these errors compound if the same area of terrain is converted multiple times. Figure 6.15 shows the result of this compounding error. However, this only becomes very noticeable when the same section of terrain has been converted between the two representations more than five times. As we can see from the figure, once the patch of terrain has been converted five times, the terrain still looks relatively realistic. Only by comparing with the original terrain section do we realise that some loss of detail, and a slight rise in the height of the area has occurred.

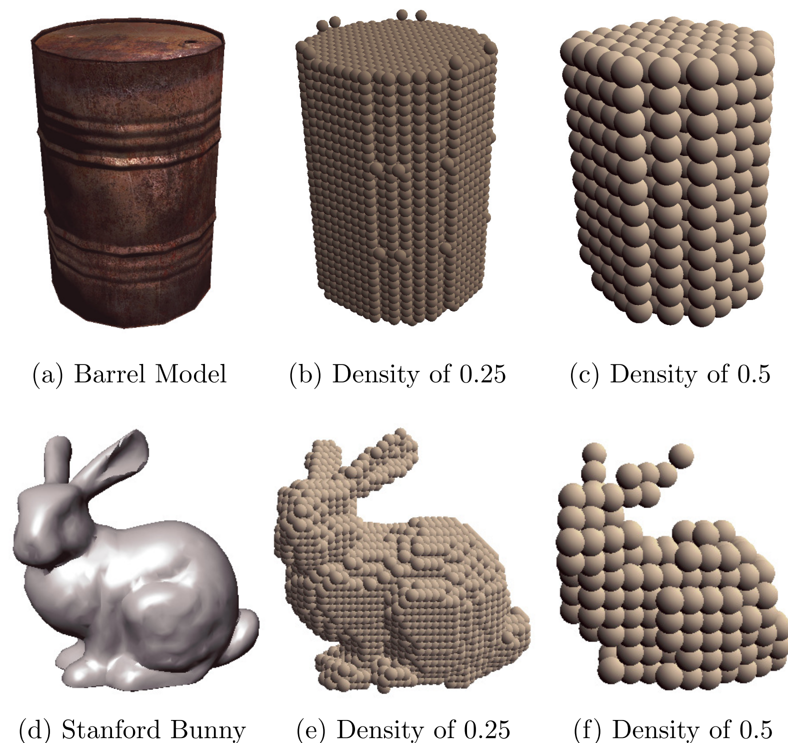


(a) (b) (c)

Figure 6.15: The error present compounds through multiple conversions. (a) Shows a patch of terrain that has yet to be converted. (b) The same patch of terrain has been converted 5 times. (c) The same patch of terrain has been converted 15 times.

Next we examine the conversion of models to the particle-based representation. Figure 6.16 shows an example of a barrel converted to the particle based representation.The particle-based representation conforms well to the base model. Indeed, looking at the results from the videos, it can be seen that the models interact realistically with the terrain. This even extends to the shadows, which appear where you would expect them to.

However, slight errors may be seen in the resulting converted models, due to very fine edges. An example of these errors may be seen in Figure 6.16b. Around the top of the barrel, there is a thin lip. In a few places the sampling density coincides with this lip, thus resulting in particles being added. However, at lower sampling densities, such as Figure 6.16c, this does not occur. Unfortunately, these artefacts are extremely difficult to combat, without introducing additional artefacts. The results, however, remain acceptable, as the resulting artefacts do not result in any strange behaviour when the object interacts with the terrain, and the particle representation is never seen by the user.



(a) Barrel Model (b) Density of 0.25 (c) Density of 0.5

(d) Stanford Bunny (e) Density of 0.25 (f) Density of 0.5

Figure 6.16: Two models are converted to a particle-based representation. (a) Is a simple barrel model. (d) Is the Stanford bunny. The particle-based representation for two different particle scales is presented, corresponding to two differently scaled particle systems.

Finally, the fidelity of the resulting terrain is examined. As can be seen from the images below, the terrain appears realistic, with smooth transitions between particle-based and heightfield-based representations. Shadows fall in a realistic way, with no noticeable discontinuity between the shadow on the terrain and the shadow on the particle system.

Figure 6.17: Example of the integrated system running, complete with shadow maps and model interactions.

Figure 6.18: Another example of the integrated system.

Observing the results though, it is quite obvious that the particle sizes are far too large to represent individual grains of sand. Fortunately, sand is not the only form of granular material. Materials such as rubble and pebbles also exhibit granular interactions, and thus are also possible candidates. An example of the system being used to simulate pebbles can be seen in Figure 6.19. We can see that this form of material is currently a much better fit for our system. However, we expect that the future advancement of GPUs will provide even greater computational resources, and thus realistically simulating sand will soon be possible.

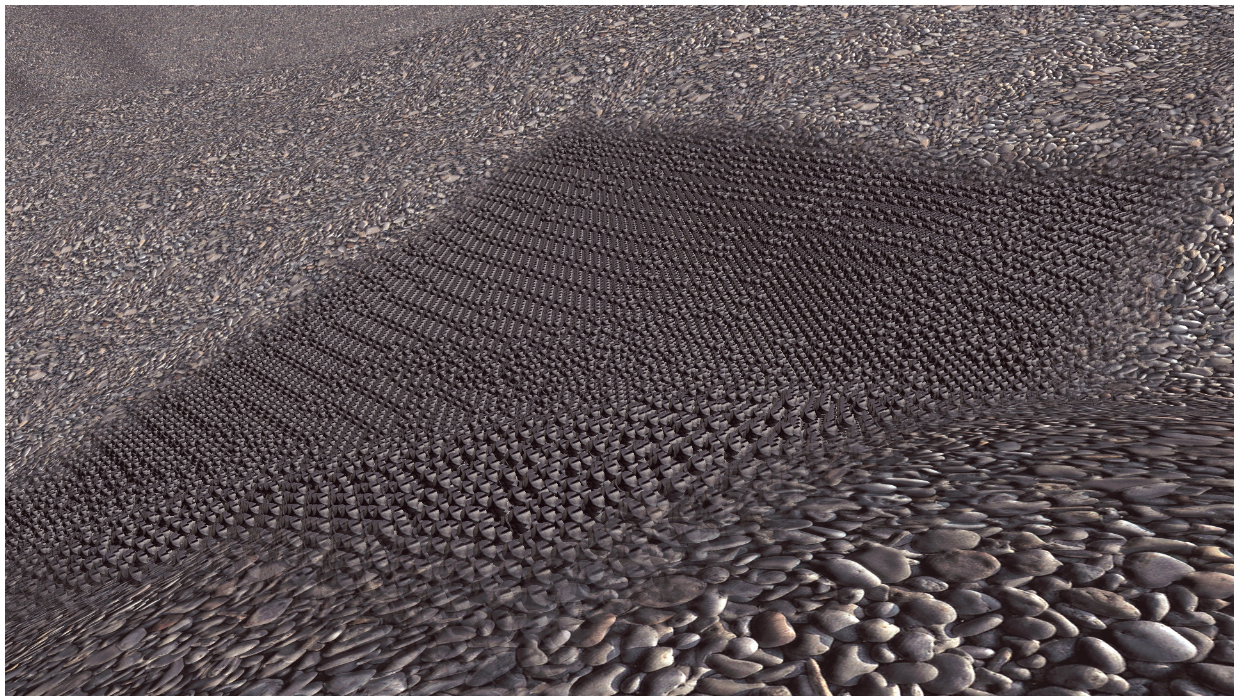


Figure 6.19: An example of the our system rendering pebbles. As we can see, the size of the particles is a good match for this form of granular material.

The sections of particle-based terrain are also rather distinct from the heightfield-based terrain. However, this problem is only visual. A rendering technique, closer to the regular grid structure of heightfield-based terrain could be employed to negate this problem. However, this is beyond the scope of this thesis, and has therefore been left as future work.

6.5 Summary

In this chapter, the results of the level of detail system were analysed. The individual system components were analysed, as well as the complete integrated system. The GPU geometry clipmap implementation, in particular, was extremely efficient, and was capable of rendering large terrains at over 750 frames per second on average. The resulting integrated system is capable of simulating large-scale granular terrains in real-time. The terrain displays realistic interactions with dynamic objects, with minimal error exhibited through terrain conversions, which means that terrain may freely be converted from one representation to the other, without undermining the integrity of the underlying terrain.

However, the system remains computationally expensive. While the more powerful nVidia GTX 770 was able to handle all of the tests without issue, the nVidia GTX 460 dropped below 30 frames per second on multiple occasions. That is not to say that the system is unusable with the card. But certain restrictions on its use are imposed by the performance, such as the density of the particle simulations, and the number of active simulations. At this point in time, the system is best suited to more powerful hardware. Fortunately, with the computational power of GPUs increasing every year, the system will soon be suitable for use with midrange consumer GPUs.

Chapter 7

Conclusion

Scenes in modern games and simulations consist of many dynamic objects, each exhibiting complex interactions with other objects in the scene. Leaves and branches sway in the wind and move in response to collisions with other objects. Crates and boxes can be stacked upon each other, or knocked over. But there is a significant disparity between the interactions exhibited by terrains and the interactions exhibited by other objects in a scene. This is particularly evident with sandy or granular terrain, which should be easily deformable, and display fine-level granular interactions. Such terrain is often represented by static geometry with no dynamic interactions, or otherwise exhibits unrealistic interactions.

Recently, particle-based granular terrain simulations have appeared as an alternative to the traditional heightfield-based terrain systems. While previously limited clusters, it has recently become possible to implement such a system on modern desktop computers. This has been made possible by the evolution of the GPU from a specialised graphics processor to a more general purpose processor. Particle-based granular terrain simulations exhibit physically correct interactions with dynamic objects and support high-fidelity rendering. However, their high computational complexity means that they may only be feasibly used for smaller areas of terrain.

In order to overcome this limitation, we have created a hybrid system, which combines a particle-based granular terrain simulation with a heightfield-based level of detail system. The resulting system is capable of simulating large-scale granular terrain in real-time, by seamlessly switching between the heightfield-based and particle-based representations, with changes to the terrain in the particle-based simulation persisting in the heightfield-based system, by virtue of a novel rendering based conversion technique.

While we originally aimed to simulate sand, this is not possible, due to the size of the particles produced by the particle-based simulation. Even the finest grained simulations resulted in particles that are much larger than individual sand granules. However, other granular materials may still be simulated, such as pebbles or rubble. Further advances in GPU processing power should allow for finer grained materials, such as sand, to be realistically simulated in the future.

7.1 Heightfield-Based Terrain

Our GPU geometry clipmaps implementation is capable of rendering large-scale terrain extremely efficiently. This is achieved by dividing up the terrain into concentric grids of increasing coarseness. This was shown to dramatically reduce the number of geometric primitives required to render the terrain, thereby increasing rendering performance. Each of these concentric grids is made up of multiple smaller grids, which further aids rendering performance, by allowing the culling of areas of terrain which fall outside of the view frustum. We made some changes to the standard implementation, such as a revised VBO structure for the grids, and a simplified method for upsampling from the coarser clipmap levels.

A novel texturing technique was also introduced. This technique, whilst simple, allows the system to efficiently texture the terrain, without the complex overhead required by previous techniques. This is achieved by effectively projecting the texture onto the surface of the terrain.

The performance of our GPU geometry clipmaps implementation is excellent. As shown in the results, the system achieves a framerate of over 750 frames per second with an nVidia GTX770. Performance is hindered somewhat by shadowing the terrain, although the framerate remains above 400 frames per second.

7.2 Particle-Based Terrain

The data structures which store the particle and rigid body attributes were analysed. We showed how the rendering based update algorithm leverages this representation to perform updates to the particles, by mapping the particles to a grid structure, which effectively allows particles to interact with each other. The rendering and shadowing techniques, which allow for efficient and realistic presentation of the particle system, were also addressed.

We also examined the scaling of the particle system, to allow support for varying particle system scales. Two potential techniques were presented. The first operates by using a smaller particle size, and a correspondingly denser collision grid. The forces calculated also had to be scaled, as the particles would get closer to one another before they collide. However, this approach was deemed inappropriate, as the scaling of the forces and constants is non-linear, and thus results in arbitrary parameter tuning. The second technique is much simpler, but proved to be more effective. This technique maintains the same particle size and collision grid density, and simply scales the rendering of the particle system. This resulted in more stable interactions, and doesn't require any arbitrary parameter tuning.

Comparing the results to those of Longmore[29], there was a dramatic increase in performance, due to the added computational resources available on newer GPUs. The particle-based simulation is capable of simulating and rendering up to 600,000 particles in real-time. Shadow mapping once again limits performance somewhat, but performance remained good with the GTX 770 still able to render over 300,000 particles in real-time.

7.3 Terrain Manager

The terrain manager we developed is able to convert between the terrain representations in real-time. Of particular importance is the top-down orthographic rendering technique, which allows for a particle system to be quickly converted to a heightfield-based representation, while ensuring that the changes made to the terrain in the particle system persist in the heightfield-based terrain. Additionally, we demonstrated that a simple Gaussian filter could be applied to remove artefacts from the resulting converted terrain region.

We also examined how the terrain manager manages the various particle systems, assigning dynamic objects to currently unused particle systems. By tracking the movement of dynamic objects within the simulation, particle systems may be disabled when objects come to rest, thus allowing for the appearance of more active particle systems than are actually present. Finally, the conversion of models to a particle-based representation was addressed.

In order to test the system, three different test scenarios were created. Each scenario tests a different facet of the system. Real-time results were recorded with a modern high-end GPU, whilst a mid-range GPU was able to perform most of the tests whilst still maintaining real-time performance. The conversion system was also tested by converting from the heightfield-based representation to the particle-based representation and back again. It was shown that a minimal error is exhibited during the conversion, and that this error is directly proportional to the size of the particles used in the particle-based simulation.

7.4 Future Work

Whilst this thesis addresses many of the issues with rendering large-scale granular terrain in real-time, many potential avenues for further research remain.

One potential avenue concerns interactions of the particle system with water. Beaches are one of the most common use cases for sandy terrains within virtual environments. However, the particle system presented only displays interactions with rigid bodies. Many particle-based water simulations exist. It may be possible to integrate such a system with the particle-based terrain system, in order to create a generalised system for simulating water, granular terrain, and the interactions between them.

Another potential extension to the particle system concerns the rendering of the particles. Whilst the splat-based rendering technique presented is extremely efficient, it results in a uniform appearance across the particle system. This is obviously contradictory to real sand, where granules within a volume of sand appear distinct from the other granules. An instance-based approach could be taken, where various different geometric representations are available, and are assigned to the particles based on their identification numbers.

Alternatively, the rendering technique could be changed to a system which more closely matches the regular grid structure of heightfields, or the particle system could be converted to a heightfield using the current method each frame.

One of the serious limitations of the system is that the particle system scale is set at instantiation, and thus the particle system cannot be refined to a finer grained simulation as the observer approaches it. This could potentially be addressed in the future, by creating a system whereby particles or granules could be split into multiple smaller versions, thereby allowing the particle system to be refined as the user approaches.

A heightfield-based interaction could be used for objects further away from the observer. This would reduce the number of particle systems required to perform interactions with the terrain, and would allow for fewer, higher-fidelity systems to be used closer to the observer.

The heightfield to particle-system conversion could be optimised in the future. Known stable configurations of particles could be injected into the system, and combined in order to potentially eliminate the settling period. This would allow the terrain to adapt more quickly to interactions with objects. Additionally, this could reduce errors caused by shifting of the particles.

Finally, the system could also be adapted to allow the particle system to always be running in an area around the observer, as they move around the terrain. This would require the ability to remove arbitrary particles from the simulation, and insert new particles as the user moves around. Note that the removal of particles from the simulation is not currently supported. Additionally, particles would have to be converted to a heightfield based representation, whenever they are removed by falling out the particle system bounds.

Chapter 8

References

[1] Bullet physics library. http://bulletphysics.org/, 2013.

[2] Akenine-Moller, T., Haines, E., and Hoffman, N. *Real-Time Rendering*, third ed. A K Peters, Ltd., 2008.

[3] Angel, E. *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*, 4th ed. Addison-Wesley Publishing Company, USA, 2006.

[4] Asirvatham, A., and Hoppe, H. Terrain rendering using GPU-based geometry clipmaps. In *GPU Gems 2* (2005), Addison-Wesley, pp. 27–45.

[5] Bell, N., Yu, Y., and Mucha, P. J. Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2005), SCA '05, ACM, pp. 77–86.

[6] Blinn, J. F. Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1978), SIGGRAPH '78, ACM, pp. 286–292.

[7] Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. Bdam batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum 22* (2003), 505–514.

[8] Cook, R. L. Shade trees. *SIGGRAPH Comput. Graph. 18*, 3 (Jan. 1984), 223–231.

[9] Corporation, N. Physx. https://developer.nvidia.com/technologies/physx, 2013.

[10] CROW, F. C. Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph. 11*, 2 (July 1977), 242–248.

[11] DE BOER, W. H. Fast Terrain Rendering Using Geometrical Mipmapping, 2000.

[12] DIMITROV, R. Cascaded shadow maps. http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf, 2007.

[13] DORJGOTOV, E., ARNS, L., AND BERTOLINE, G. Bertoline g.: Granular material interactive manipulation: Touching sand with haptic feedback. In *In Proceedings of the 14-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* (2006), pp. 295–304.

[14] DUCHAINEAU, M., WOLINSKY, M., SIGETI, D. E., MILLER, M. C., ALDRICH, C., AND MINEEV-WEINSTEIN, M. B. Roaming terrain: real-time optimally adapting meshes. In *Proceedings of the 8th conference on Visualization '97* (Los Alamitos, CA, USA, 1997), VIS '97, IEEE Computer Society Press, pp. 81–88.

[15] ERLEBEN, K., AND DOHLMANN, H. Signed distance fields using single-pass gpu scan conversion of tetrahedra. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, 2008, pp. 741–763.

[16] GARLAND, M., AND HECKBERT, P. S. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., pp. 209–216.

[17] GROUP, K. OpenCL. http://www.khronos.org/opencl/, 2013.

[18] GUMHOLD, S., WANG, X., AND MACLEOD, R. Feature extraction from point clouds. In *In Proceedings of the 10 th International Meshing Roundtable* (2001), pp. 293–305.

[19] HAMMERSTONE, R., CRAIGHEAD, M., AND AKELEY, K. Vertex buffer object specification *Revision 0.99.6.* http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt, 2010.

[20] HARADA, T. Real-time rigid body simulation on gpus. *GPU Gems 3* (2007), 123–148.

[21] Hasenfratz, J.-M., Lapierre, M., Holzschuch, N., Sillion, F., GRAVIR, A., et al. A survey of real-time soft shadows algorithms. In *Computer Graphics Forum* (2003), vol. 22, Wiley Online Library, pp. 753–774.

[22] Hoppe, H. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 99–108.

[23] Hoppe, H. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Visualization '98. Proceedings* (1998), pp. 35–42.

[24] Hwa, L. M., Duchaineau, M. A., and Joy, K. I. Adaptive 4-8 texture hierarchies. In *Proceedings of the conference on Visualization '04* (Washington, DC, USA, 2004), VIS '04, IEEE Computer Society, pp. 219–226.

[25] Jones, M. W., Brentzen, J. A., and Sramek, M. 3d distance fields: A survey of techniques and applications. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS 12* (2006), 581–599.

[26] Larsen, B. D., and Christensen, N. J. Real-time terrain rendering using smooth hardware optimized level of detail. *Journal of WSCG 11*, 2 (feb 2003), 282 – 289. WSCG'2003: 11th International Conference in Central Europe on Computer Graphics, Visualization and Digital Interactive Media.

[27] Lengyel, E. *Mathematics for 3D Game Programming and Computer Graphics*, third ed. Charles River Media, 2012.

[28] Li, X., and Moshell, J. M. Modeling soil: Realtime dynamic models for soil slippage and manipulation. In *In Computer Graphics Proceedings, Annual Conference Series* (1993), pp. 361–368.

[29] Longmore. Towards realistic interactive sand: A gpu-based framework. Master's thesis, University of Cape Town, 2009.

[30] Longmore, J.-P., Marais, P., and Kuttel, M. Towards realistic and interactive sand simulation: A gpu-based framework. *Powder Technology* (2012).

[31] Losasso, F., and Hoppe, H. Geometry clipmaps: terrain rendering using nested regular grids. In *ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), SIGGRAPH '04, ACM, pp. 769–776.

[32] Ltd, T. R. Havok physics. http://www.havok.com/products/physics, 2013.

[33] Luebke, D., Watson, B., Cohen, J. D., Reddy, M., and Varshney, A. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[34] Maciel, P. W., and Shirley, P. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 symposium on Interactive 3D graphics* (1995), ACM, pp. 95–102.

[35] Meehan, M., Insko, B., Whitton, M., and Brooks Jr, F. P. Physiological measures of presence in stressful virtual environments. In *ACM Transactions on Graphics (TOG)* (2002), vol. 21, ACM, pp. 645–652.

[36] Nguyen, H. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[37] Nouguier, C., Bohatier, C., Moreau, J. J., and Radjai, F. Force fluctuations in a pushed granular material. *Granular Matter 2* (2000), 171–178. 10.1007/PL00010912.

[38] NVIDIA Corporation. Cuda. http://www.nvidia.com/object/cuda_home_new.html, 2013.

[39] O'Brien, D., Fisher, S., and Lin, M. C. Automatic simplification of particle system dynamics. In *In Computer Animation* (2001), pp. 210–218.

[40] Peuker, T. K., Fowler, R. J., Little, J. J., and Mark, D. M. The triangulated irregular network. In *In Proceedings DTM Symposium American Society of Photogrammatry - American Congress on Survery and Mapping* (1978), pp. 24–31.

[41] Pla-Castells, M., Garca-Fernndez, I., and Martnez, R. Interactive terrain simulation and force distribution models in sand piles. In *Cellular Automata*, S. El Yacoubi, B. Chopard, and S. Bandini, Eds., vol. 4173 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 392–401. 10.1007/11861201_46.

[42] Pla-Castells, M., García-Fernandez, I., Martinez-Dura, R. J., et al. Physically-based interactive sand simulation. *Eurographics 2008-Short Papers* (2008), 21–24.

[43] Rosenthal, P., and Linsen, L. Smooth surface extraction from unstructured point-based volume data using pdes. *IEEE Trans. Vis. Comput. Graph. 14*, 6 (2008), 1531–1546.

[44] Rusinkiewicz, S., and Levoy, M. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), ACM Press/Addison-Wesley Publishing Co., pp. 343–352.

[45] Schneider, J., and Westermann, R. GPU-friendly high-quality terrain rendering. In *The 14th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2006 (WSCG 2006)* (Bory, Czech Republic, 2006), vol. 14.

[46] Shapiro, L. G., and Stockman, G. C. *Computer Vision.* Prentice Hall, Englewood-Cliffs NJ, 2001.

[47] Sigg, C., Peikert, R., and Gross, M. Signed distance transform using graphics hardware. In *Visualization, 2003. VIS 2003. IEEE* (2003), IEEE, pp. 83–90.

[48] Smith, R. Open dynamics engine. `http://www.ode.org/`, 2007.

[49] Solenthaler, B., and Gross, M. Two-scale particle simulation. In *ACM Transactions on Graphics (TOG)* (2011), vol. 30, ACM, p. 81.

[50] Tanner, C. C., Migdal, C. J., and Jones, M. T. The clipmap: a virtual mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 151–158.

[51] Torchelsen, R., Comba, J., and Bastos, R. Practical geometry clipmaps for rendering terrains in computer games. In *Shader X6 – Advanced Rendering Techniques* (2008), Charles River Media, pp. 103–114.

[52] Valient, M. Stable rendering of cascaded shadow maps. *Shader X6 – Advanced Rendering Techniques* (2008), 231–238.

[53] Westover, L. Interactive volume rendering. In *Proceedings of the 1989 Chapel Hill workshop on Volume visualization* (1989), ACM, pp. 9–16.

[54] Williams, L. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph. 12*, 3 (Aug. 1978), 270–274.

[55] Williams, L. Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1983), SIGGRAPH '83, ACM, pp. 1–11.

[56] Zach, C. Integration of geomorphing into level of detail management for realtime rendering. In *Proceedings of the 18th Spring Conference on Computer Graphics* (New York, NY, USA, 2002), SCCG '02, ACM, pp. 115–122.

[57] Zeng, Y.-L., Tan, C. I., Tai, W.-K., Yang, M.-T., Chiang, C.-C., and Chang, C.-C. A momentum-based deformation system for granular material. *Comput. Animat. Virtual Worlds 18* (September 2007), 289–300.

[58] Zhang, F., Sun, H., Xu, L., and Lun, L. K. Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications* (New York, NY, USA, 2006), VRCIA '06, ACM, pp. 311–318.

Appendix A

Videos

Whilst static images are useful for giving an idea of how a system works, they can only convey so much information. With real-time applications, it is advantageous to see the system in action, to get a better idea of how it performs in a general use case. In order to show the system in use, we have created four videos, which are available online.

General Overview http://youtu.be/uE2o-gnyhKA
: This video shows the conversion of the terrain from the heightfield-based representation to the particle-based representation and back again. It also shows objects interacting with the terrain, including multiple objects interacting with each other in one of the scenes.

Test Scenario 1 http://youtu.be/Cnskfn8FQKA
: The first test scenario. See Section 6.4.1 for more information.

Test Scenario 2 http://youtu.be/wfU7V3KZon0
: The second test scenario. See Section 6.4.2 for more information.

Test Scenario 3 http://youtu.be/gjepmHkNr4k
: The third test scenario. See Section 6.4.3 for more information.

All videos were recorded, and have been uploaded at a resolution of 1920×1080. Note that there is an overhead incurred by the recording process, so the framerate is slightly lower than in actual use.

Appendix B

Comparison of Terrains With and Without GPU Geometry Clipmaps

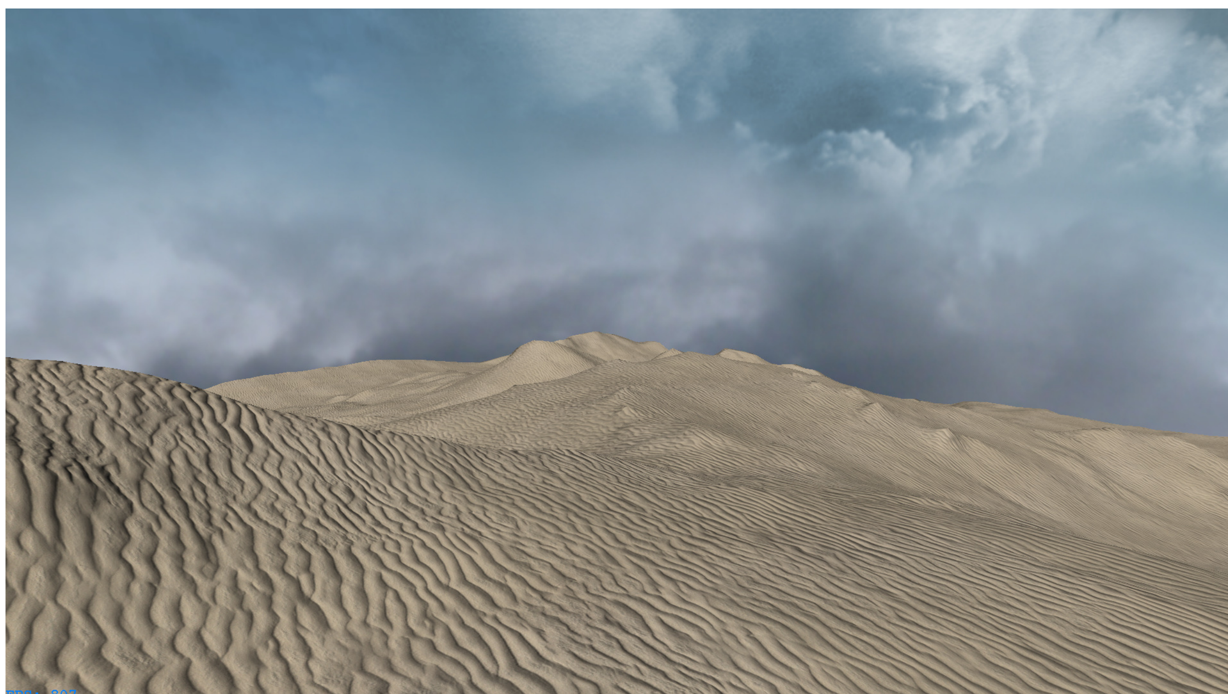
It is often difficult to quantitatively ascertain the loss of image quality when using a level of detail technique. A common method is to compare the pixel values between images, in order to measure the percentage of different pixels between the base terrain, and the level of detail technique. However, this can be misleading. For instance, differences in the foreground are far more noticeable than differences further from the camera, but using this direct comparison, they are assigned equal weights.

Thus, in order to show the visual accuracy of our GPU geometry clipmaps implementation, we have provided screen shots of terrains rendered with and without GPU geometry clipmaps. This will allow you to judge the quality of the resulting terrains for yourself. Four different terrains are used, with each terrain rendered from exactly the same viewpoint in each case. The images can be found on the following pages. The frames per second are shown in the bottom left of each image, to show that the LOD system is indeed being used where stated.

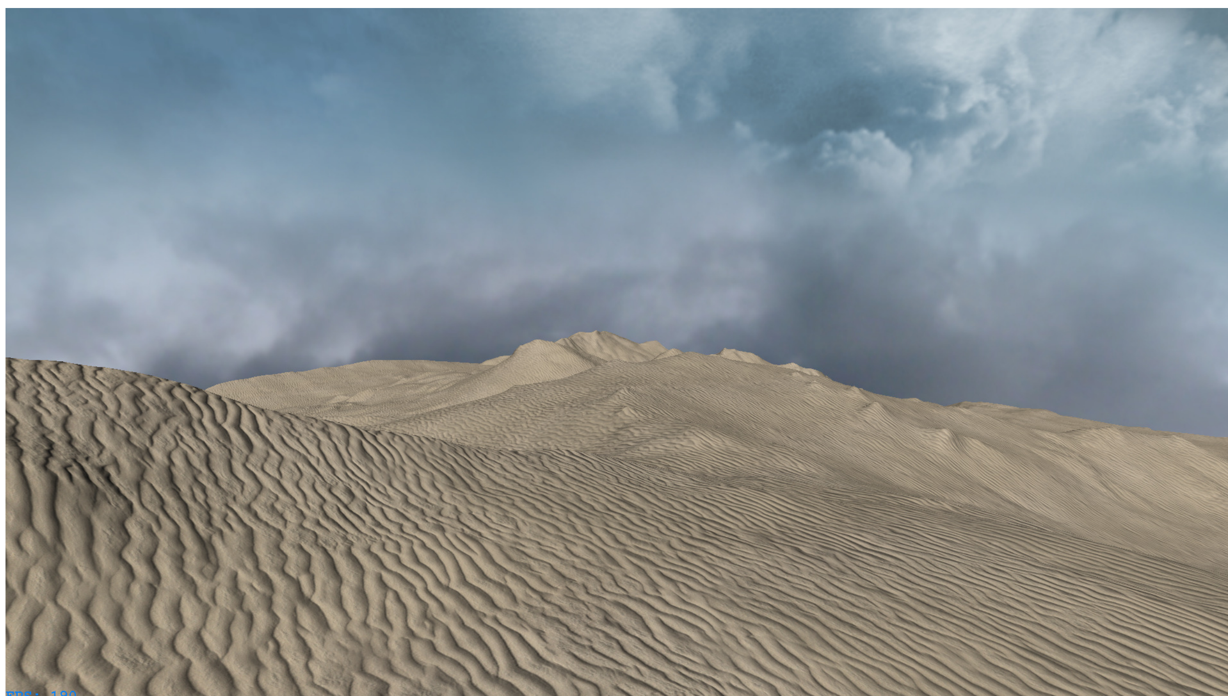
As we can see from the images, while there are certainly differences between the terrains presented, these differences are minor, and the terrain still appears realistic. This is due in part to the fact that most of the differences lie far away from the camera, where the difference in quality is less noticeable. Additionally, there is a noticeable increase in performance when using the GPU geometry clipmaps technique.

(a) With Geometry Clipmaps (779 FPS)

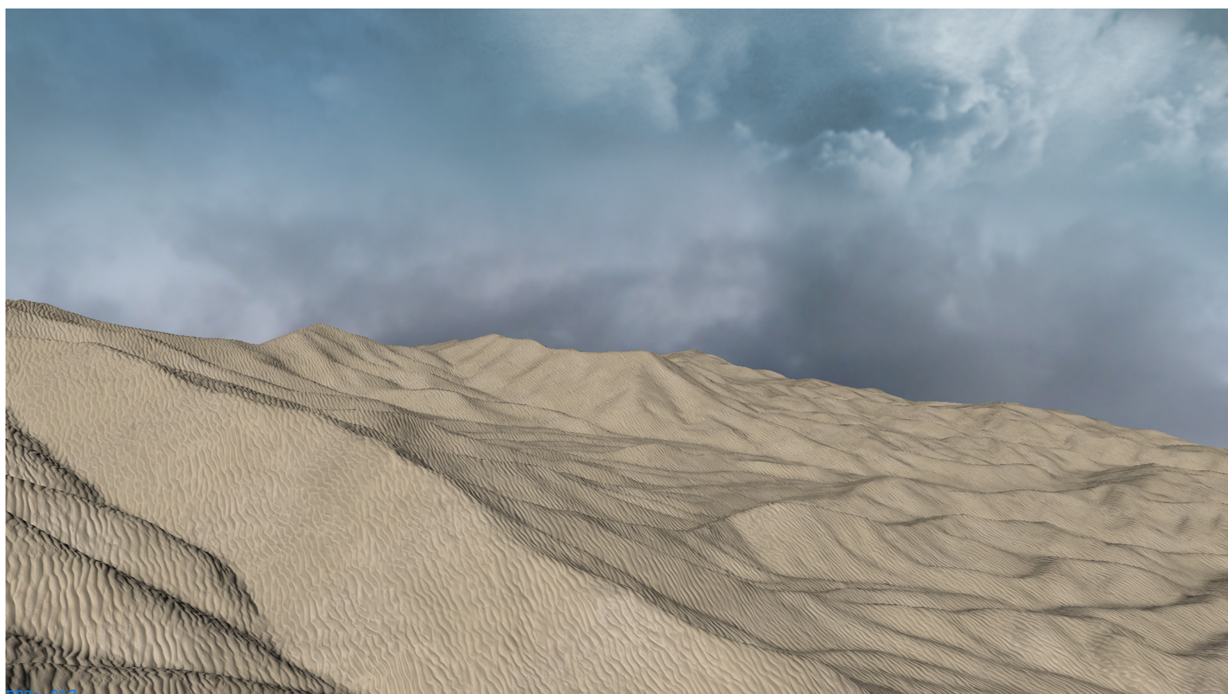
(b) Without Geometry Clipmaps (192 FPS)



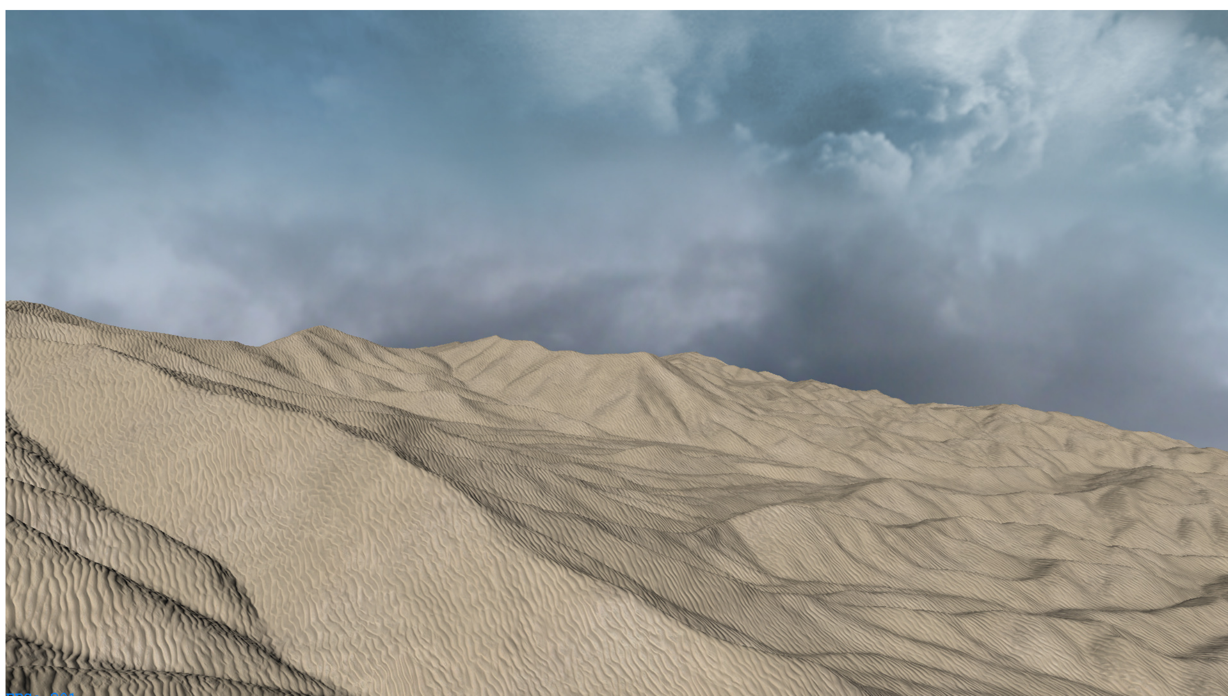
(c) With Geometry Clipmaps (807 FPS)



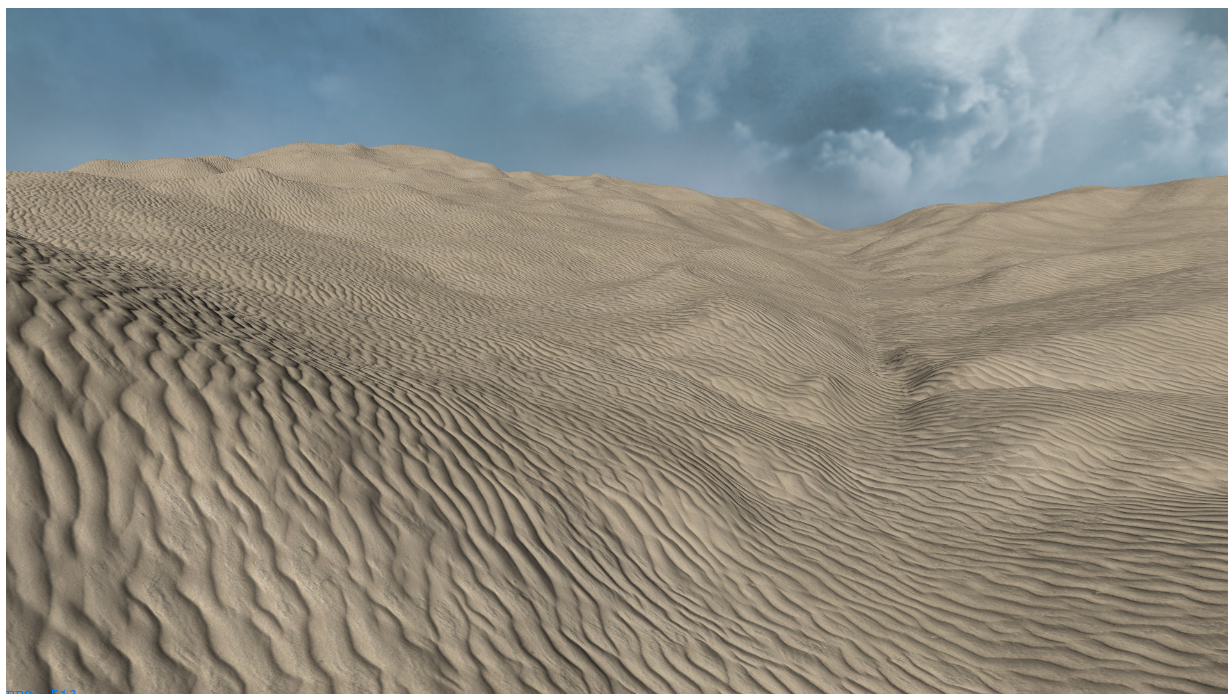
(d) Without Geometry Clipmaps (190 FPS)



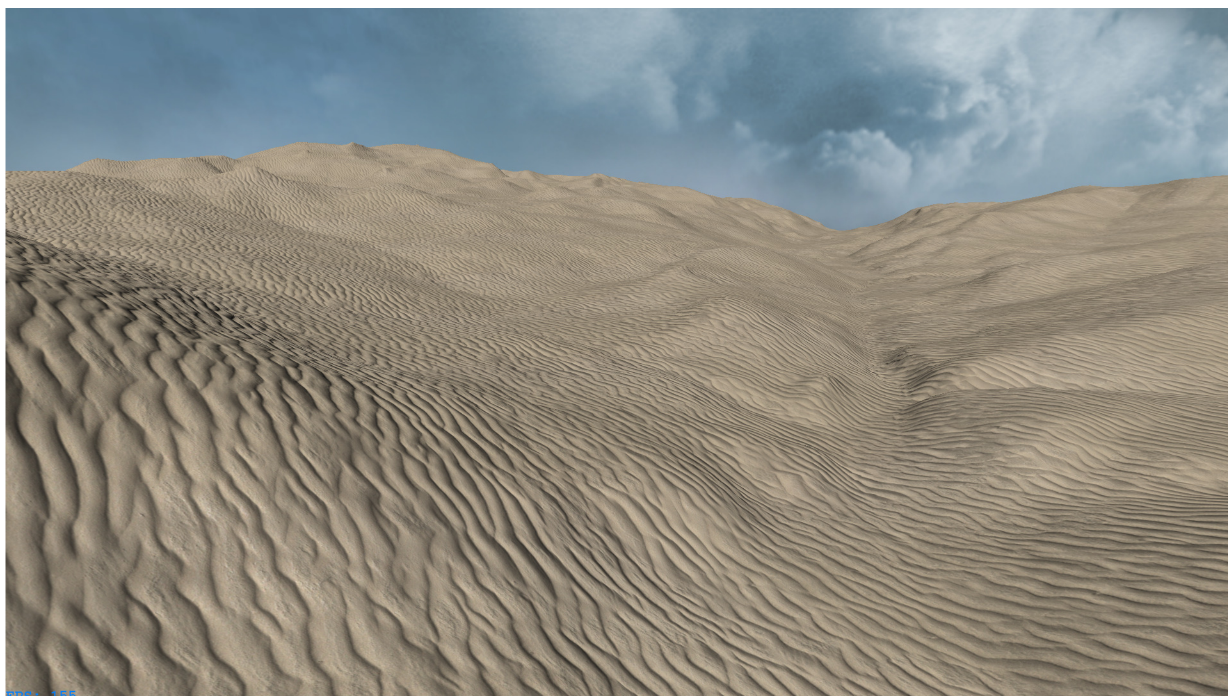
(e) With Geometry Clipmaps (917 FPS)



(f) Without Geometry Clipmaps (201 FPS)



(g) With Geometry Clipmaps (513 FPS)



(h) Without Geometry Clipmaps (155 FPS)