Saimaa University of Applied Sciences
Faculty of Technology Lappeenranta
Information Technology

Simo Huhtiranta

Approach to deformable terrain visualization and a soil behaviour model

Bachelor's Thesis 2015

Abstract

Simo Huhtiranta
Approach to deformable terrain visualisation and a soil behaviour model, 47 pages
Saimaa University of Applied Sciences
Information Technology Lappeenranta
Degree Programme in Information Technology
Bachelor's Thesis 2015
Instructor: Mr Pasi Juvonen, D.Sc.

The objective of this work was to research and develop the visualisation of deformable terrain and a soil behaviour model in a real-time 3D simulation software.

This work was done as a part of Mevea software development project, which covered the design and implementation of a deformable terrain extension to Mevea simulation software. This work discusses the initial design of the terrain extension and addresses the problems that were discovered during the design and implementation process.

The resulting document discusses the problems of deformable terrain visualisation and presents an approach that was used in the implementation of Mevea simulation software.

Keywords: deformable terrain, real-time simulation

Contents

1 Introduction **5**
- 1.1 Overview . . . 5
- 1.2 About Mevea . . . 5
- 1.3 Motivation . . . 6
- 1.4 Improvement over existing system . . . 6

2 Terrain simulation as a concept **7**
- 2.1 Terrain simulation . . . 7
  - 2.1.1 Simulation primer . . . 7
  - 2.1.2 Simulation model . . . 8
- 2.2 Example usage scenario . . . 8
  - 2.2.1 Value for end users . . . 9
- 2.3 Functional requirements . . . 10

3 Mevea simulation software **12**
- 3.1 Software components . . . 12
  - 3.1.1 Solver . . . 12
  - 3.1.2 Simulation model . . . 12
  - 3.1.3 Modeller . . . 13
  - 3.1.4 Visual channel . . . 13
- 3.2 Terms and concepts . . . 14
  - 3.2.1 Graphics . . . 14
  - 3.2.2 Cave . . . 14

4 System environment considerations **14**
- 4.1 CPU . . . 15
- 4.2 GPU . . . 15
- 4.3 Network bandwidth . . . 17

5 Terrain extension design **17**
- 5.1 Technical requirements . . . 17
  - 5.1.1 Terrain model . . . 18
  - 5.1.2 Materials . . . 18
  - 5.1.3 Mixing materials . . . 18
  - 5.1.4 Terrain Object . . . 18
  - 5.1.5 Soil particle . . . 19
  - 5.1.6 Merging particles . . . 19
  - 5.1.7 Synchronisation . . . 19
  - 5.1.8 Rendering . . . 19
- 5.2 Elevation data . . . 20
  - 5.2.1 Triangulated irregular network . . . 20
  - 5.2.2 Heightmap . . . 21
- 5.3 Extension structure . . . 22
- 5.4 Data storage . . . 23
  - 5.4.1 Grid data . . . 23
  - 5.4.2 Terrain model . . . 25

5.4.3 Render component 26

6 Terrain rendering **28**
6.1 OpenGL 28
6.1.1 Primitives 29
6.1.2 Shaders 29
6.1.3 Shader inputs 29
6.1.4 Deferred shading 30
6.2 Material blending 30
6.2.1 Blend operator 30
6.3 Rendering strategies 32
6.4 Final terrain texture 33
6.4.1 Blending in fragment shader 33
6.4.2 Using accumulation strategy 35
6.4.3 Virtual texture approach 35
6.5 The terrain geometry 36
6.5.1 Simple quads approach 36
6.5.2 Optimised terrain geometry 36
6.6 Rendering terrain particles 37
6.6.1 Particle shader 37
6.6.2 Particle groups 37

7 Results **40**
7.1 Example simulator using the terrain extension 40
7.2 Performance 41

8 Summary **41**
8.1 Learned techniques and methodologies 42
8.2 Algorithm and software related knowledge 43
8.3 GPU debugging 43
8.4 Conclusion 44

List of Figures **45**

References **46**

Figure 1. A forwarder simulated in forest environment (Mevea Ltd.)

1 Introduction

1.1 Overview

The goal of this work was to design and to implement a terrain visualisation extension to Mevea simulation software. This paper outlines the initial design of the *terrain extension*, reviews obtained results and discusses the potential future development of the terrain visualisation used in Mevea simulators. This work focuses more closely on the visualisation module of the said extension, its data back-end, and data sources. This work will not cover how the soil model dynamics are realised.

C++ and OpenGL technologies were used in the implementation.

1.2 About Mevea

This work is part of an ongoing development of Mevea simulation software. Mevea Ltd is a software company that develops and sells real-time simulators and associated simulation software. The produced simulators are often used to complement the training of the operators of different sorts of working machinery. The machinery to which this simulation technology is applied ranges from concrete spraying vehicles used in mining tunnels to large freight container cranes working in cargo ports. These simulators are collectively referred to as *training simulators*. Mevea also provides *R&D simulators* for the purpose of developing entirely new machine hardware by using simulation as a research tool. Both types of simulators may feature dynamic and hydraulic simulation models used in real-time calculation but in addition to those, the training simulators also feature rich visualisation to create immersive experience for the user.

1.3 Motivation

Mevea had a need to extend its software — mainly the training simulators — with the ability to operate simulated vehicles on soft terrain. This feature had been frequently requested by customers and it was decided that it would become part of Mevea software package (see Section 3). The feature would be realised as a set of components which together would form what is here referred to as the *terrain extension*.

1.4 Improvement over existing system

Prior to this extension the terrains used in Mevea simulators were defined using static geometry. This geometry had the visual appearance of a terrain but lacked the distinctive behaviour of a real soil. These static terrains were used in rigid body simulation but they were unyielding objects that did not have the ability to deform under forces applied to them.

For many simulator applications this had been enough. A lot of simulated vehicles such as mining or cargo machinery did not require the deformable terrain feature since they operate on solid ground.

There are many situations however where static geometry does not provide sufficient realism to the simulation. A good example is a forest harvester that is most often operated on soft forest floor. The soft terrain can be muddy and very difficult to drive on. This challenging environment should be correctly simulated in order to provide a level of realism that closely reflects the real machine behaviour.

The *terrain extension* was designed to complement the feature set offered by *Mevea simulation software* to better meet the requirements of the customers and to allow new kinds of simulators to be developed.

2 Terrain simulation as a concept

This chapter discusses the functionality of the new terrain extension in a broader context. This chapter answers the questions:

- What does terrain simulation mean?
- Who would want to use it and why?
- What value will it create to its users?
- What are the functional requirements of the extension?

2.1 Terrain simulation

In a broad sense *terrain simulation* is an idea of how to capture the behaviour and look of a real terrain in a computer program. Terrain simulation can be defined in many ways depending on which aspects of terrain are relevant to the problem at hand. The following chapter will discuss the various aspects of real world terrain and how they relate to the concept of terrain simulation. Also the scope of the simulation will be defined.

2.1.1 Simulation primer

In this work the term *terrain* is used to mean both the surface and volume of earth.

A real world terrain is a complex structure of different shapes, materials and formations that have been formed through many ongoing geological processes for the last four and a half billion years. These processes include the movement of tectonic plates, volcanic eruptions and various erosion effects carried out by wind, rain and ice. The phenomena range from the slowest to blazingly fast and from the very subtle to violently fierce. These processes — although very different in behaviour — do essentially the same thing: they move matter. This happens across the surface, through the air and throughout the Earth's volume. These processes change the structure of Earth which — in the most abstract sense — is the core concept of what terrain simulation is all about.

2.1.2 Simulation model

The simulation of terrain is accomplished through a *simulation model*. A simulation model is a collection of data structures and functionality that together mimic certain aspects of a real terrain.

Considering the vast diversity and the infinite detail present in the real world it can easily be understood that it is not possible to develop an ideal simulation model able to address all the properties and phenomena of the ground. Rather than attempting the impossible, the simulation model must be constructed to address a specific problem in a specific problem domain.

This work focuses on a particular area of the terrain that lies closest to the human perspective. The research questions were:

1. How to make the terrain interact with machine wheels and tracks?
2. How to use these machines to move earth from one place to another?
3. How is it possible to work with heterogeneous soil with many different materials?

The scope of the simulation model is narrowed down to address only the questions above. This means all phenomena not covered by the scope will be excluded entirely, such as the phenomenon of erosion. The preconditions and requirements implied by these questions are further discussed in Chapter 2.3.

Pla-Castells, García-Fernández and Martínez (2006) have done previous research on the subject of real-time simulation models of granular systems. (1, 2)

2.2 Example usage scenario

A simple use case story describes the usage and context of the *terrain extension*.

An earthmoving machine manufacturer decides to use a Mevea simulator to promote its product in an exhibition to attract new customers. The simulator is constructed to mimic a certain excavator model of the manufacturer and is fitted with authentic controls used in the excavator model in question. People trying out the simulator are presented with a task of building a ramp.

A CAD design tool is used to define the environment terrain and the shape of the ramp. The slope and shape of the ramp are precisely defined because they will

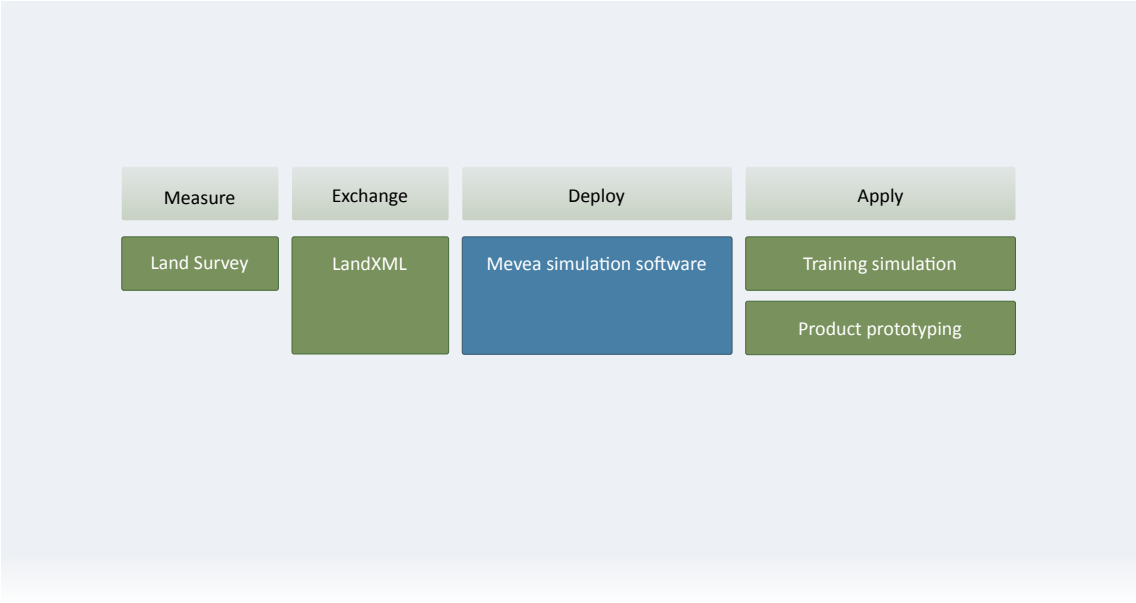


Figure 2. Extension concept

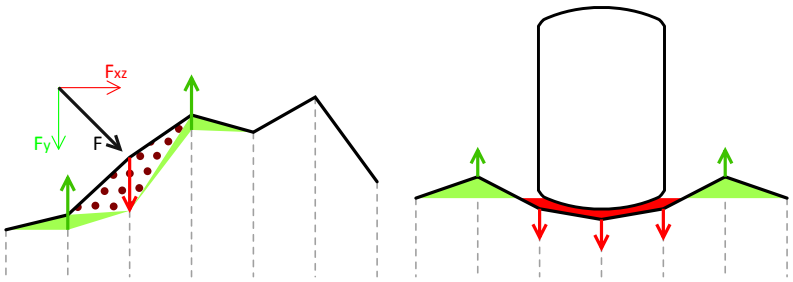
be used as a foundation for a road. The surrounding environment in the simulator is a replica of a real world location measured with a *LIDAR* technology. LIDAR—Light Detection and Ranging—is a remote sensing method used to examine the surface of the Earth (3).

The people attempting the task use the simulator controls to move land from a dedicated place to a target site. The simulator exhibits a 3D GPS system that displays the complete ramp shape at all times so that people can see where to deposit the material. When a person completes the task they are presented with their success rate and statistical information of their achievement.

2.2.1 Value for end users

In this example the terrain extension makes it possible for the client to showcase their product and to provide training in a simulated environment where the conditions demand soft soil simulation. It allows their customers to interact with their machine in a controlled and safe environment while using terrain simulation to produce realistic behaviour comparable to that of a real machine.

Sometimes the client might want to feature a real landscape location in a simulator application. For example, they could want to simulate an earthmoving task in a replicated environment to see how to accomplish it most efficiently. Perhaps they would like to compare various excavator models to perform the same task or to see how different machines work on the task together.



(a) Horizontal force generates particles. Vertical force displaces soil.

(b) Land compression and displacement

Figure 3. Different phenomena of ground

The possibility to use data from a real landscape location is addressed in the overall concept of the terrain extension. This is noted by allowing clients to provide their own data for the simulation model. The extension is destined to support the LandXML format for importing terrain data from real world locations. LandXML is a supported format in many CAD engineering tools that can be used to exchange the required data (4). This allows the client to use their tool of choice to produce the data that meets their needs and to transfer it to the Mevea simulator. Figure 2 illustrates a concept of the phases involved in deploying the LandXML technology.

2.3 Functional requirements

The functional requirements describe how the extended system should behave and what actions can be performed by it. The identification of the functional requirements is necessary because the requirements form the basis for the design and implementation phases of the work. The requirements are drawn from the usage scenario described above and also from interviews with customer companies.

The primary requirements are:

1. The simulated ground must have the behaviour of a soft soil.
2. There must be a way to pick up soil material and deposit it to a different location.
3. There must be different soil materials present in the simulation and these have to be mutually distinguishable.
4. The soil and the *deformation* of the soil must be visualised and should look

appealing.

The first requirement means the ground which the simulated vehicle is driven on exhibits the properties of compressibility and deformability (see Figure 3b). When a wheel or a track is in contact with the ground, the system formed by these engaging elements should exhibit the same behaviour and appearance as they would in the real world. This is one of the most important features of the extension and reflects the fundamental philosophy of Mevea, which is to accomplish the most realistic simulation available.

The second requirement states that it must be possible to separate selected portions of soil which can then be manipulated as independent entities. After the manipulation it must also be possible to merge these portions back into the ground when they are no longer needed. This requirement stems directly from the use case of different earthmoving machines as it enables the feature of digging earth.

The third requirement states that there must be more than one material in the simulated world and these materials should all have their own appearance and dynamic properties. While this is quite straightforward and clear statement it is not entirely clear how it should be implemented and where the requirement stands in relation to the other requirements. When considering requirement 2 — a case where a material is being deposited on top of a different material — there exists an *implied* requirement that the terrain model should allow some sort of mix between the materials. Also there is an implied requirement that the materials should form some sort of structure, such as sediment layers.

The fourth requirement speaks for the importance of the visualisation and relates to all of the other requirements. Visualisation is an essential part of the simulation experience and will often determine whether the simulation is deemed realistic and adequate. The fourth requirement states that not only the terrain mass but also its *movement* should be visualised. This implies the presence of ground particles (see Figure 3a) which are used to simulate and visualise the movement of soil. This is an essential feature of the soil model that enables the simulation of landslides and digging of the ground.

These are the four primary requirements that must be addressed by the design and the implementation of the terrain extension. They are discussed more thoroughly in the requirement analysis in Chapter 5.

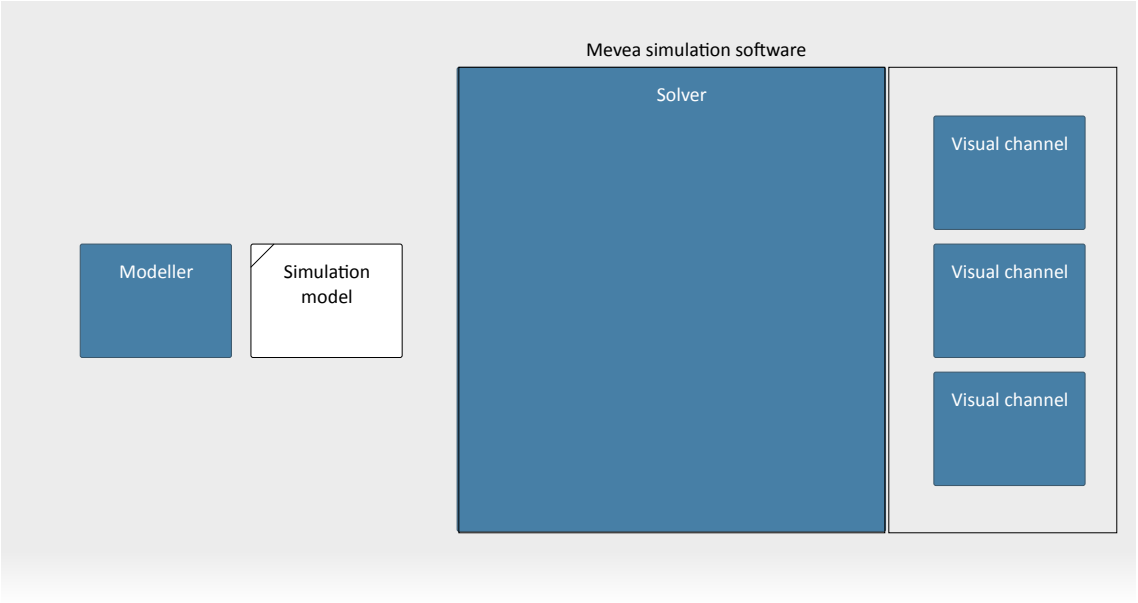


Figure 4. Simulator software components

3 Mevea simulation software

3.1 Software components

Mevea simulation software as a whole is a development platform upon which many kinds of simulators and simulations can be built. It is a set of tools for running simulations, building the *simulation model* and integrating the simulator with hardware components such as *motion platforms* and controllers. Some concepts of the simulation software must be explained in order to understand the whole concept of the terrain extension.

3.1.1 Solver

The *Solver* is a component of the simulation software that performs the dynamic simulation calculation. It accounts for most of the workload of the simulator as it *solves* rigid body, soft body and hydraulic simulation systems in real-time. These dynamic systems are defined by a *simulation model.*

3.1.2 Simulation model

A *simulation model* combines all data needed to run a simulation. It is a configuration that defines the system of dynamic *bodies*, their properties and their relations

between each other. It also references the *graphics* used to visualise most of the components inside the simulation. All of the components present in the *simulation model* define the starting state of a simulation.

3.1.3 Modeller

The modeller is the editor for building *simulation models*. With the modeller it is possible to define the components (hydraulic or dynamic), their properties and their relation to other components. Some of the basic properties of these components are mass information (magnitude and centre) and inertia tensors. The modeller is also used to define the look of the visualised components. This can be done by assigning *graphics* and *shaders* to components.

A *simulation model* of a wheel loader, for example, could be built by these basic steps:

- defining *dynamic bodies*
- assigning *graphics* to the bodies
- defining hydraulic components such as cylinders
- adding a power source (diesel motor in this example)
- defining inputs to control the model at runtime

After these steps the simulation model can be read by the solver and executed. The wheel loader may then be driven assuming all appropriate inputs for steering and other controls have been defined.

3.1.4 Visual channel

Visual channel is a software component that allows the solver to delegate the visualisation of the simulation to one or more computers across network. This relieves the *solver node* (the central computer) from the visualisation overhead and enables more external views into the simulated world than would otherwise be possible with a single computer alone (see Figure 5). The factors that limit the number of views include the number of GPUs in the system and the complexity of the visualised items drawn on screen. Visual channels are used in multi-monitor or c*ave* set-ups that typically use three or more projectors to draw each wall of the cave.

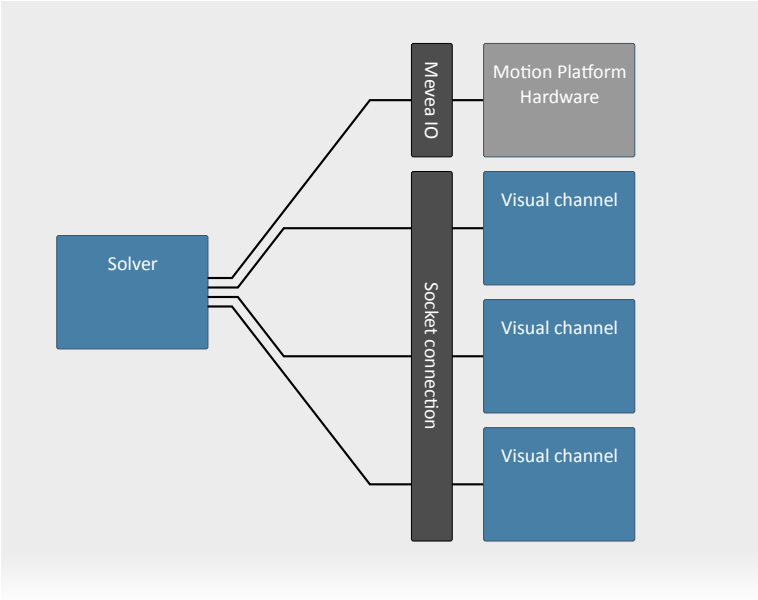


Figure 5. Hardware topology

3.2 Terms and concepts

3.2.1 Graphics

In Mevea simulators a *graphic* is a context dependent term used to refer to 2D images and 3D models collectively. These include 3D models of machine parts, user interface icons and hydraulic symbols. Other items may also fall under this term.

3.2.2 Cave

A cave is a setup of screens that enclose the viewing person inside so that they are able to view the simulated world by looking around. The screens are typically connected to visual channels.

Lin, Pan, Yang and Shi (2002) have done research on a similar kind of network-based cave system. (5)

4 System environment considerations

When designing the structure and functionality of the extension it is critical to pay special attention to the system specific limitations and preconditions. In the case

of real-time simulation there are many hard requirements that must be considered when designing each component of the system and the system as a whole. These technical boundaries often limit the design choices available and frequently dictate the direction to which the design must be taken. The system limitations are of direct consequence of the availability of system resources, which are especially scarce in real-time simulation systems. In this case the most substantial environmental limits are (in order of significance):

1. CPU time (cycles)
2. GPU time and memory
3. network bandwidth

These limitations are present in the system and must be taken into account in the design process. What their effects on the system are and what can be done to overcome them is discussed next.

4.1 CPU

CPU cycles are often the most revered system resource because there always seems to be shortage of them. Everything not run on the GPU or other processing hardware will be run on the CPU. Currently the solver executes most of its dynamic simulation on the CPU, which can — depending on the simulation model — create a substantial load for the whole system. For this reason all components developed for the simulator software should minimise the need for heavy calculation by using suitable algorithms for their task or by delegating the processing to the graphics processing unit. Sometimes a good solution does not exist for a processor intensive task, which means the task must be run in the hard way on the CPU.

4.2 GPU

GPU handles the rendering of the 3D graphics and post processing effects. The limits related to GPU are most often the limits introduced by the graphics pipeline, which in this case is OpenGL. (6) The limiting factors for GPU are most often related to the following factors:

1. shader complexity

2. number of states (API calls) needed to draw an entity

3. number of vertices

The usual bottleneck in graphics pipeline is at the fragment stage. This is the stage where the colour of each individual pixel or sub-pixel is calculated. The factors that determine the computational load at the fragment stage are shader complexity and number of pixels drawn. These depend directly on what is being drawn, what shaders are used, what textures are used, how closely the drawn entity is being inspected (perspective matters) and how much overdraw is happening during the draw procedure. This issue can be amended by using optimised shaders and adjusting the order in which objects are drawn. All solid surfaces should be drawn in order from close to far. This maximises the benefit of z-buffering, which means that the shading calculations of *fragments* can be skipped if they are behind already drawn fragments.

The other significant factor that may cause rendering slowdown is the number of graphics API calls (GL calls in this case) that the renderer has to make in order to draw an object. The API calls are needed for operations like switching shaders, changing shader *uniform* values and streaming textures. It is not required to issue every API call at every *frame*, some GL calls are only needed for start-up operations such as copying static object geometry to *graphics memory* and defining the shaders used to render it with. However, the calls used to change the GL states for rendering purposes are needed at every frame. These calls are used for selecting the object geometry buffers, switching on the correct texture units and issuing the final draw call. The number of calls needed depends on the structure of the rendered geometry and the structure of the scene. A sound strategy for rendering geometry is to minimise the need for state changes by grouping the geometries using the same set of GL states together (material colour for example) and then draw these geometries in a batch. Another way to reduce the number of calls would obviously be reducing the complexity of the drawn geometry.

The third factor that affects rendering efficiency is the number of vertices drawn. This lays burden on the vertex transform stage of the rendering pipeline and can also affect the amount of overdraw later in the fragment stage. Large geometries need equally large vertex buffers in the graphics memory and they are likely to lengthen the start-up time of the simulator, because larger objects need proportionally longer processing times. This issue can be addressed by using smaller geometries or by using modern GPU capabilities such as tessellation shaders.

The GPU is a powerful processing device which can sometimes be used for lifting

some of the load away from the CPU. Unfortunately this is often difficult if the problem in question does not translate well to the parallel GPU architecture.

4.3 Network bandwidth

Network bandwidth can become an issue when external PCs are used for visualising the simulation in a multi-monitor setup. The visual data is sent to the renderer PCs via visual channels (see Figure 5). If the network traffic in the visual channels becomes too heavy it will be seen as lag in the movement of the simulated objects. This issue is very rare however as the biggest bottleneck most often is the dynamic simulation.

The visual channels use a LAN network connection so the theoretical upper bound of the bandwidth is one gigabit per second as defined by the Gigabit Ethernet specification. If this value (10^9 bits) is divided by the required frame rate (60 frames per second) the maximum amount of data emitted per frame would be roughly two megabytes. The data which is sent to visual channels must respect this limit.

5 Terrain extension design

This chapter discusses the design of the terrain extension and aims to provide answers to the following questions.

- What are the technical requirements of the extended system?
- What are the designs and techniques needed to realise the extension?
- What are the limiting factors and technical boundaries of the environment and how have these been considered in the design?

5.1 Technical requirements

This section aims to identify the technical requirements that must be met to implement the terrain extension. The technical requirements are derived from the functional requirements introduced in Section 2.3.

5.1.1 Terrain model

There shall be a *terrain model* that defines the data of a large terrain mass. The terrain model shall have information of elevation and soil materials that are part of the terrain. The terrain model must be able to contain at least four different materials.

5.1.2 Materials

A material definition describes the visual and physical properties of a substance. The information stored in each material definition will include physical information used in dynamics simulation and bitmap images used in visualisation.

The physical information included in a material definition shall include at least density, compressibility and friction coefficient.

Material substances have two distinct forms in the simulator. Material can be in soil and particle form. A material must include the required images to describe its appearance in both of these forms.

5.1.3 Mixing materials

There should be a way to mix and blend different materials. It must be possible to mix at least two materials at any point of the terrain surface.

If possible, the materials should mix while in a particle form too. This implies that the physical coefficients will be also mixed. The result of this mix may not yield correct or even usable values for dynamics so this should be regarded as an experimental feature.

5.1.4 Terrain Object

There shall be a *Terrain Object* that is used to access and modify terrain data. The terrain object shall provide interface to read and write terrain data. The interface allows access to all terrain data or specified part of it.

The terrain object acts as a view to the terrain model. The terrain data shall only be accessed through this object in the simulator. The terrain object may provide

several views to the terrain model at the same time but these regions must not overlap.

5.1.5 Soil particle

The solver generates soil particles when objects or forces interact with the simulated terrain in such a way that causes the soil material to separate from the ground. The generated particles will represent the amount and type of material that was separated from the point of extraction. There must be a way to sample a material from any point of the terrain surface and assign it to each particle extracted. The sampled material may be a single material or a mix of several materials if applicable. The particle then carries information about this material and can deliver it to other places.

5.1.6 Merging particles

The solver may decide to merge soil particles back into terrain. The terrain object shall handle the mixing of materials during these merge operations when two or more materials are present at the merge point.

5.1.7 Synchronisation

There must be a possibility to synchronise two or more terrain objects with a single “master” terrain object. The synchronisation shall be done via a socket connection.

The synchronisation will be unidirectional. The terrain object at the receiving end may reside on the same PC as the dynamic solver (a single PC setup) or at a visual channel node (multi-screen setup). The receiver terrain object will be accessed only by the render engine so only the visual information of the terrain model needs to be relayed to the receiver object.

5.1.8 Rendering

There shall be a renderer component for drawing the terrain and soil particles. The renderer must be able to blend at least two materials at each point of the terrain surface. The renderer must be able to draw at least 2000 soil particles during one frame. Soil displacement should be visualised if possible.

A sophisticated rendering technique should be considered if possible. The rendering of the terrain should be optimised by reducing the drawn geometry complexity in the places that are occluded or lie far away from the viewpoint.

The minimum allowed frame rate is 60 frames per second.

5.2 Elevation data

Terrain simulation and rendering requires a lot of data that needs to be available in a usable format. The choice of supported data formats, in particular the format used within the simulator, greatly determine what the technical requirements of the system are. The data format has an impact on how to store, process, visualise and produce such data and what tools are needed to do so. This chapter will discuss the benefits and advantages of different data formats from the perspective of real time simulation with a strong bias on the efficiency of the whole simulator system.

The most important and perhaps obvious type of data is terrain surface elevation, which is required in one form or another. Terrain elevation in *GIS* (geographical information science) can be represented in various data structures, some of which are better suited to real time simulation than others. Common vector based formats used in GIS include terrain contour lines and *TIN* (triangulated irregular network). In addition to vector based formats the elevation can be encoded into a raster image, also known as a *heightmap*. All formats have their intrinsic properties that have significance when considering the applicability of the format in a real time simulation.

Two formats were considered to be the extension surface base formats. The properties and behaviours of the formats have to be compared from the perspective of the solver and visualisation to determine which would better serve its purpose in the simulation.

5.2.1 Triangulated irregular network

Triangulated irregular network (TIN) is a three-dimensional data structure that is used to represent surface data. It consists of elevation points connected with non-intersecting lines which together form a surface of triangles. TINs are a form of vector-based digital geographic data and are constructed by triangulating a set of vertices (points) (7).

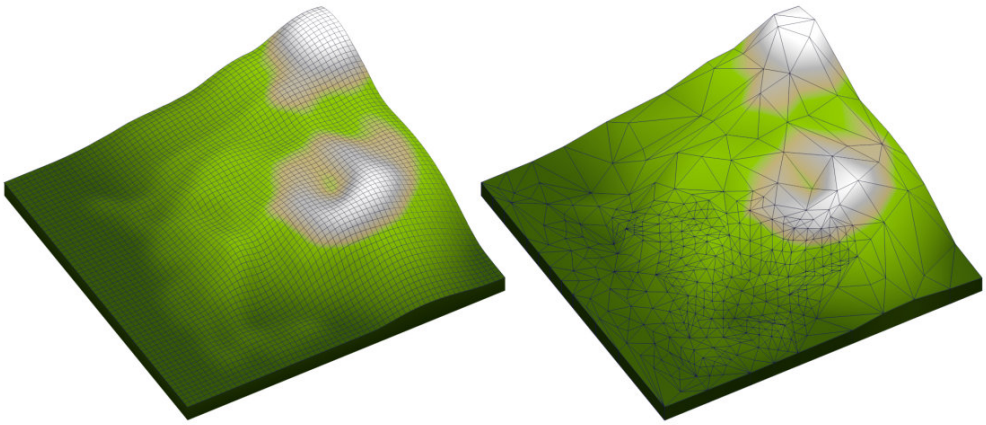


Figure 6. Heightmap and TIN with roughly similar memory requirements

A TIN is a compact data structure whose memory requirement depend on the complexity of the terrain it is representing. A TIN surface can be refined in areas where more data points are required to more accurately represent detailed geographic features, while it also may be simplified in places with less detail by using larger triangles and less data points.

A TIN surface is able to represent complex terrain formations which would not otherwise be possible with 2D data structures. These formations include cavities and overlapping terrain masses, such as steep mountain sides with protruding faces. In this regard a TIN surface surpasses two-dimensional data structures such a *heightmap*.

5.2.2 Heightmap

A *heightmap* is a two-dimensional regular array used to store elevation data. The data it stores can be understood as a matrix of elevation points that have constant distance between each neighbouring point. Unlike TIN, which stores three values (x,y,z) for each point, a heightmap needs only to store one — the elevation value — as the two other coordinates of the point can be calculated from its array index. However, depending on the resolution and size of the heightmap it may require a large amount of memory to describe a terrain.

Concerning memory consumption it is difficult to compare these two approaches as their memory requirements are essentially implementation dependent. For example, a TIN has to store triangle connectivity data whose type and size depend on the implementation. The data types used in either approach can also vary

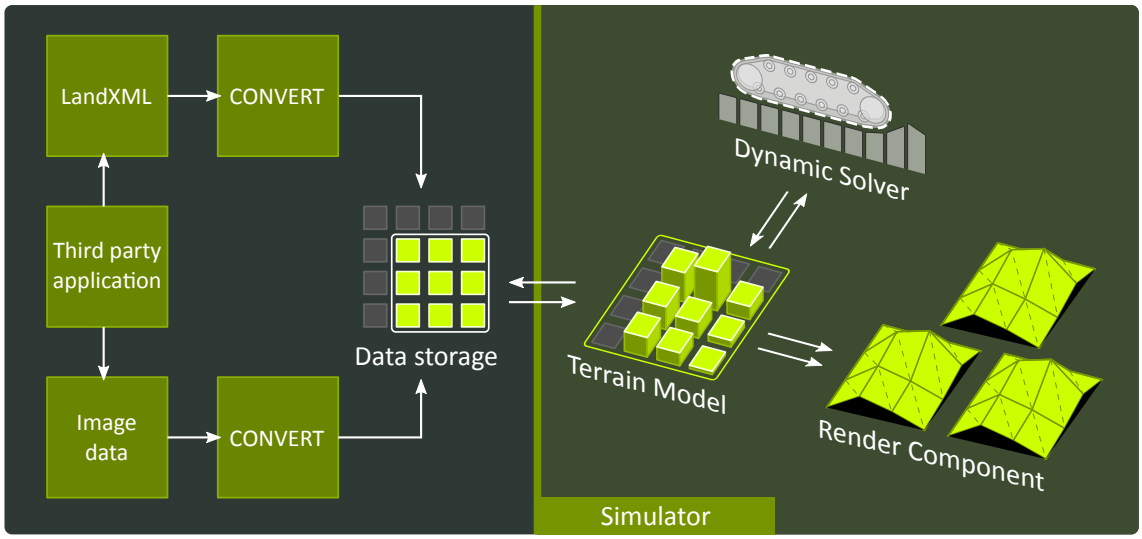


Figure 7. Module overview

between applications. For these reasons TIN and heightmap cannot be fairly compared regarding memory consumption without concrete examples of both cases. Figure 6 illustrates both a heightmap and a TIN whose memory requirements are roughly similar under a test setup.

The heightmap has two strong points when considering real-time simulation. Firstly, it is a very simple and fast data format. Its data can be allocated into consecutive memory locations, allowing fast *random access* to it. Because of this memory layout the heightmap is also likely to benefit from hardware features such as *cache buffers* and *prefetch*.

Secondly, the heightmap has a complexity which remains constant over time and space. This is an advantage over the TIN for example. A TIN surface could potentially have places where the density and shape of the mesh would slow down or even stall real-time simulation. Not only that, but consider a *deformable* TIN, where its complexity would depend on the operations performed on it. That is to say, a very simple TIN mesh could — during simulation — potentially *become* a very complex one, by introducing detail into initially non-detailed areas. A heightmap on the other hand always has the same amount of data and a constant topology, making it a more stable option for real-time simulation.

5.3 Extension structure

The extension introduces three main functions as illustrated in figure 7.

- Data storage

- Terrain model
- Render component

Each function is a logical grouping of software components that are responsible for one main task. These functions aim to match the requirements stated in Section 5.1. The design of each function is explained to give some insight of how they compose the overall extension.

The logical layout of the terrain extension design resembles the model-view-controller design pattern. Here the terrain model, render component and solver being the model, view and controller respectively. (8)

5.4 Data storage

The *data storage module* was designed to be an extensible and versatile interface to grid based data used in the simulation software. The main features of the *data interface* are:

- ability to store unbounded uniform grid based data
- ability for client code to specify the stored data type
- ability to access arbitrary sized region of data
- ability to create named layers for different types of data

The idea behind this component was to allow large terrain data sets to be specified and to access only those parts of the data that are needed at a certain time during the simulation. The data interface was designed primarily for terrain data storage but it does not depend on other simulation software components and the data types used by it are not in any way terrain data specific. This separation was an intentional design choice to make the data module easier to integrate to other supporting tools and possibly third party applications outside the simulation software.

5.4.1 Grid data

The data stored by the module can be thought to be a sparse unbounded two-dimensional regular layered grid of data. In practice the grid is not unbounded but rather the referable coordinates span from 0 to $2^{32}-1$ for both axes. For reference,

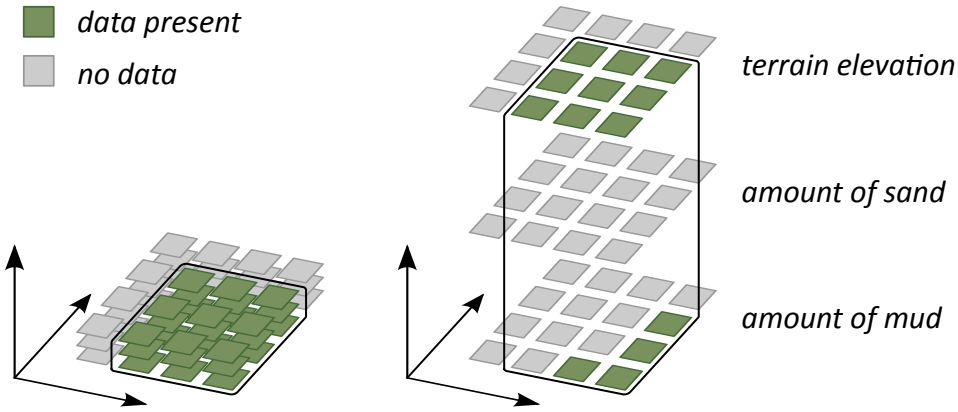


Figure 8. Compound data type and data separated into layers

the circumference of the Earth is roughly 40000 kilometres. If this length was divided to 2^{32} equal spans the resolution of the line would be roughly one centimetre. This was considered to be good enough for the purpose the module was primarily designed for.

In reality the number of referable elements in the grid is too large to fit in any hard drive regardless of the data type. For this reason the data stored in the grid is sparse in nature. This means that the user has the ability to store data selectively to only those coordinates that are relevant to the problem at hand. This is a very simple and effective way of storing moderate sized data sets. If very large data sets are needed then another storing strategy is required.

The data module allows its users to specify the types of data to store in the grid. A data type can be any *C data type* (char, integer, float, double, etc.) or it can be any combination of these. The data type must be of a constant size and its size must be a multiple of the base machine type, which is assumed to be one byte (8 bits). The user who defines a new data type must provide the functions to serialise and deserialise the data element to and from the disk. A terrain module could for example define a data type that is a combination of terrain elevation (a float) and soil colour (3 times char). The data module would then use this definition to provide the read and write functions for this type.

The data module allows the data to be stored as layers. For example, the simulation application might require data about the quantity of different materials present in the ground. The data for these materials could be stored in a compound layer including information about all materials or the information could be separated into distinct layers. If there were three materials then the compound layer would store three values (i.e. floats) at each surface point whereas the distinct layers would be three float layers.

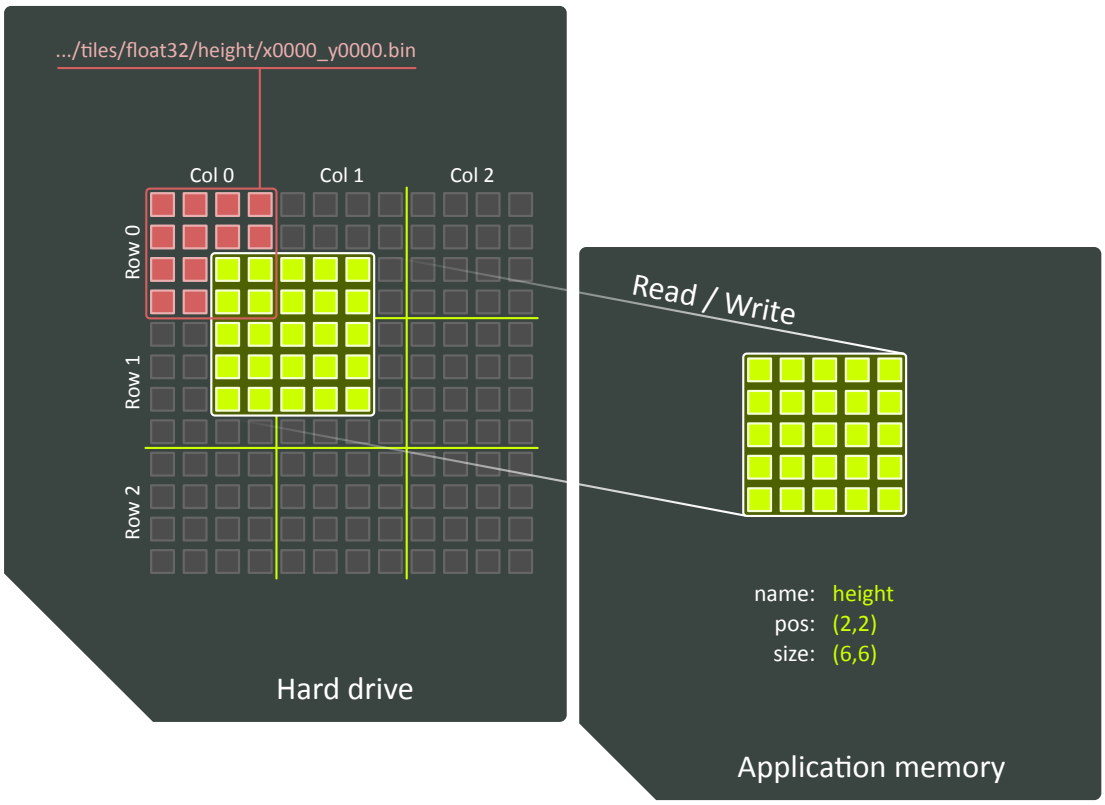


Figure 9. Data storage

If the data of each soil type was separated into distinct layers then the data for each soil type would be mutually independent. An area on the grid could for example store the amount of sand but the same area would not necessarily need to store the amount of mud if there was none. This way of storing the data requires considerably less space (if a certain material is absent) compared to the case where the information of the materials are combined into a single compound layer. Depending on the case however the compound layer could be more efficient or more straightforward to use. For this reason the data grid was designed to work with both approaches. This is illustrated in Figure 8.

The data grid implementation stores each layer as binary tiles on the hard drive or other block device. This mapping is done behind the interface and is transparent to the user (see Figure 9).

5.4.2 Terrain model

The terrain model is a concept which defines what a terrain is. It defines the materials and geometry of the terrain which is being manipulated through simulation. In this design the terrain is a simplified flat projection of data that describes the

elevation, soil type and bedrock height.

The terrain model software component is responsible for providing the means to access and manipulate the data contained within the terrain. It provides interfaces for the dynamic solver and the rendering component to access only the data required by these components while abstracting away other data not relevant to them.

The terrain component is by design an integral part of the simulator software. This close coupling is required to be able to integrate the component with the dynamic solver and to implement the synchronisation routines more effectively.

Initially it was decided that the terrain model would use a flat projection of terrain data for simplicity and efficiency reasons. This restricts the use cases of the terrain model somewhat. With flat projection it is not possible to model very large terrains that would include planetary geometry. This simplification however makes the calculation and terrain definition straightforward and is sufficient for all use cases the terrain model was primarily designed for.

The interfaces provided by the terrain model component consist of two interfaces used by the dynamic solver and rendering component respectively. The dynamic solver is only interested in the terrain surface elevation and material physical properties while the rendering component needs only to know the information about the appearance and amounts of different materials present.

The terrain model component is instantiated by two modules within the simulator. The solver node has the master terrain model which is used in the soil deformation simulation. The master terrain model has the authoritative terrain data and it handles the mixing of materials. Each visual channel has a client terrain model which are synchronised with the master terrain using the simulator event system. The master and client terrain model instances must be initialised with the same configuration and initial data set to be able to work together within the cluster. The main system ensures that the initial terrain data and configuration match before the simulation is started. The synchronisation is illustrated in Figure 10 on page 27.

5.4.3 Render component

The rendering component handles the drawing of the terrain data provided by the terrain model. The rendering component also draws the soil particles that were

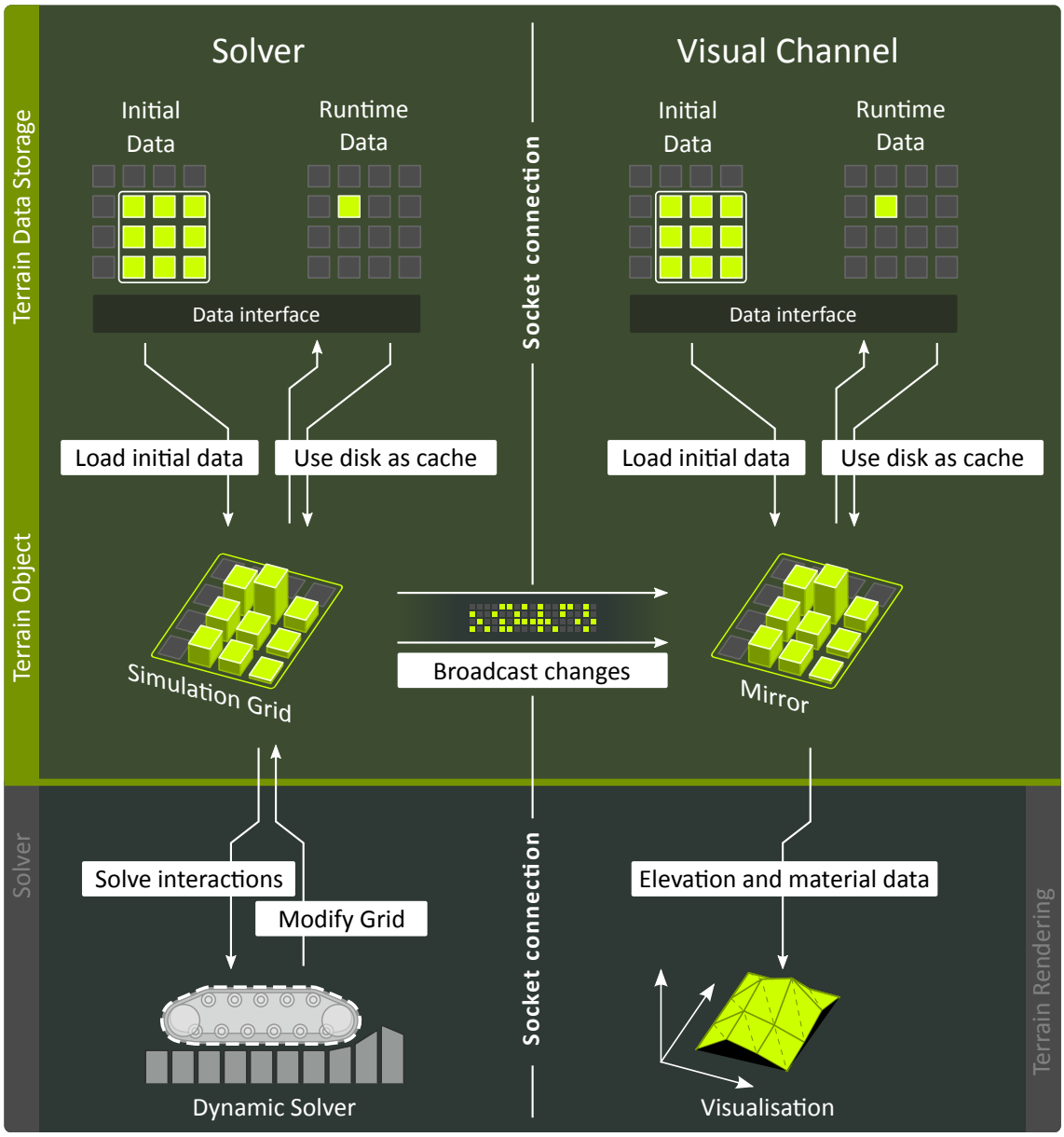


Figure 10. Data flow during simulation

detached from the terrain by the dynamic solver.

The rendering component needs the elevation data of the terrain, the visual properties of present materials as well as the quantities of these materials at any point on the terrain surface. The terrain model provides an interface to access this information.

6 Terrain rendering

The rendering of a terrain is a classic problem in the domain of real-time 3D graphics. Various techniques have been developed to make this rendering efficient in respect to frame rate and memory consumption (9–11). In this work the problem of terrain rendering can be split into a set of sub-problems.

- How to draw a large scale terrain efficiently?
- How to achieve the blending of many materials?

The rendering of the terrain needs to be efficient. This is a hard requirement which needs a sophisticated solution. In this work the problem includes the classic issues of terrain rendering as well as the problems brought by the dynamic nature of the terrain model. The data that has to be rendered is large in size as well as in diversity. The terrain has to support rapid data updates that should not affect the performance of the solution. This section discusses some techniques of terrain rendering and evaluates their applicability to meet the ends of the terrain extension.

6.1 OpenGL

The terrain is rendered using OpenGL technology. At the lowest level of terrain rendering is the OpenGL interface. This interface defines the usable primitives, data structures and operation that are used to draw the terrain on screen. OpenGL uses GLSL shaders provided with proper inputs to perform rendering operations. These rendering operations are affected by OpenGL states that define render targets as well as blending parameters.

The OpenGL API and GLSL are defined in OpenGL specification and GLSL specification respectively. (12, 13)

6.1.1 Primitives

An OpenGL based system draws most geometry on screen using three basic primitive types. These primitive types are points, lines and triangles. In addition to these, OpenGL includes other primitive types such as the triangle strip which is a more terse representation of triangles. These extended types are drawn the same way however as the three basic types.

6.1.2 Shaders

GLSL shaders are simple programs that control various stages of the OpenGL pipeline. Shaders can be used to control vertex and fragment stages that together control the shape of the primitives drawn on screen and their shading respectively. Other shader types are available for other uses but this document mainly discusses the two mentioned above.

6.1.3 Shader inputs

Input data needs to be provided to the GLSL shaders in order to render the terrain. The input data can be generally provided in two forms.

First there are the vertex attributes. A vertex attribute is a piece of data that is defined for each rendered vertex of a primitive. If the primitive is a triangle then the triangle may have vertex attributes defined for all its corners. Generally this data can be a float, a vector or a matrix. Data such as terrain elevation, colour, texture coordinates and surface normals can be provided to a shader encoded into vertex attributes.

A shader may also take its inputs as textures. The OpenGL specification defines various texture types, of which the two dimensional texture is the most common. Textures are mainly used to contain image data but they are frequently used to contain other types of data as well. The textures can contain different data types with up to four texture components per texel (r,g,b,a). A texel stands for *texture element* which is practically equivalent to a pixel in an image. The data format of each texel component can be an integer of a floating point type. Matrix data can be encoded into textures by using multiple texels or textures to encode the matrix vectors.

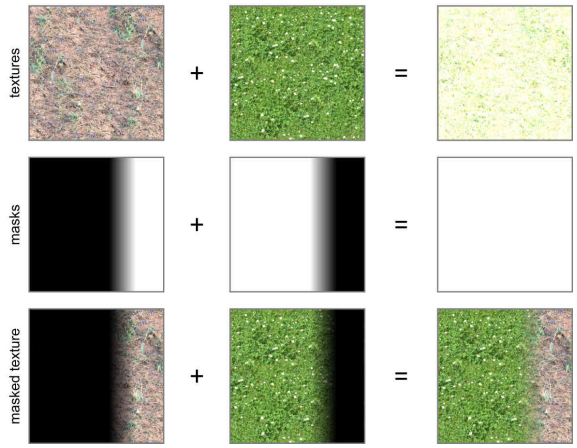


Figure 11. Blending two textures to get a mixed texture as a result

Whether to use a vertex attribute or a texture depends on the data and rendering strategy being deployed.

6.1.4 Deferred shading

Deferred shading is a concept in which the rendered primitives are not directly rendered on screen but rather to a texture or another buffer for later processing. Deferred shading can be used to perform lighting calculations as a separate rendering step to increase performance. (14)

6.2 Material blending

The terrain extension needs to be able to draw areas of the terrain surface with different soil materials. The visual properties of a material can be defined in many ways, the most simple definition probably being a single colour. A texture or multiple textures may be used to describe the material but regardless of the material visual format they will be blended using the same approach which is discussed below.

6.2.1 Blend operator

Blending is achieved through a *blend operator*. The blend operator is used to blend material properties at any point on the terrain surface with multiple materials present. The blend operator is an abstraction that describes the mathematical formula and it can be implemented as a single GLSL shader or a combination of

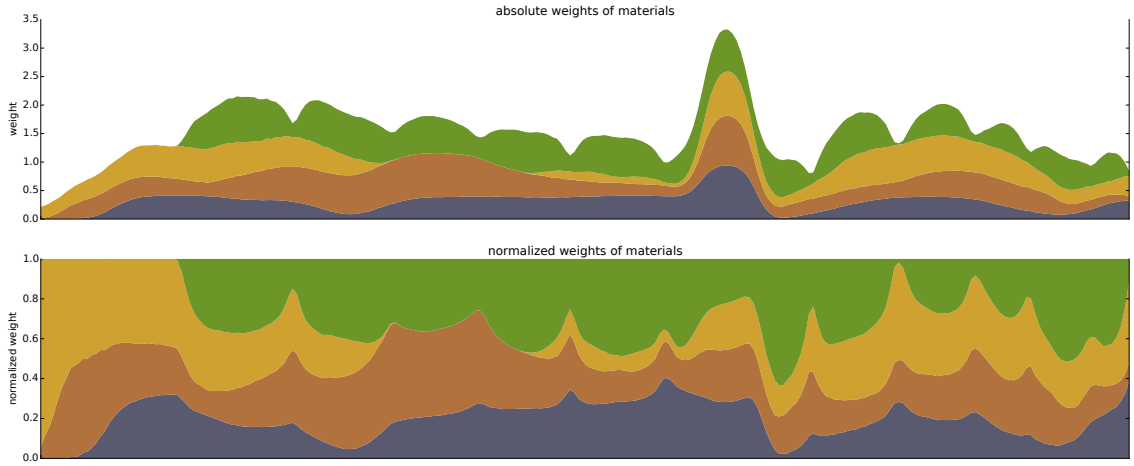


Figure 12. Normalising weights

many shaders and GL operations. How and where this operator is used depends on the deployed rendering strategy.

The blend operator is a mathematical operator for mixing two or more inputs, each of which have a weight value that determines how much the respective input operand contributes to the final output value. The equation of the blend operator is

$$C_{out} = \sum_{n=1}^{N} C_n W_n \tag{1}$$

for normalised weight inputs and

$$C_{out} = \frac{\sum_{n=1}^{N} C_n W_n}{\sum_{n=1}^{N} W_n} \tag{2}$$

for unnormalised weight inputs, where N is the number of inputs to blend, C is an input value and W is the weight value for this input. The weight operands W for equation (2) must not sum to zero as this would cause a division by zero error and yield an undefined result.

The input operand C of the blend operator can be colour data, in which case the operator will yield a blended colour. In this colour the blend operator was applied element wise to all of its colours components (red, green, blue and alpha).

(a) Without blend map

(b) With blend map

(c) Blend map for cobblestones

Figure 13. Weighted blending with and without a blend map

A simple example of a blend operator using colour textures as input operands is demonstrated in Figure 11 on page 30. In this example the equation (1) was used because the weight operands (the masks) were already normalised.

Figure 13 on the other hand shows the result of equation (2) which uses multiple texture images with varying weight inputs. The weight of the cobblestone texture is defined by a blend map (Figure 13c). The effect of the blend map can be seen in the sharp appearance of the stone shapes in Figure 13b. Figure 13a shows the result without the blend map.

Figure 12 demonstrates the effect of normalising a cross section of weight operands such as those that were used in Figure 13.

6.3 Rendering strategies

When drawing the terrain various rendering strategies can be used to render the final visual representation of the terrain surface. For the purpose of this discussion a distinction is made between the rendered geometry and the terrain texture. These two concepts are defined as follows:

The rendered geometry is defined to be the primitives (triangles) making up the final geometrical representation of the terrain surface drawn on screen.

The terrain texture is defined to be the final visual representation of the terrain colour, as defined by the lighting model, shading operators, textures and post processing effects that may be used to get the result.

In essence the terrain texture defines the look of the terrain while the geometry defines the shape. The problem of blending the material textures should be considered a problem within the domain of the final terrain texture and should not depend on drawn terrain geometry. The problem of optimised terrain geometry however lies within the domain of final terrain geometry.

6.4 Final terrain texture

One of the most challenging problems of calculating the final terrain texture is the material blending (see Section 6.2). This is because there may be a lot of materials present in the terrain with many sets of data in each material. This document assumes that each material has four textures: diffuse, specular, normal and blend map. In order to output the desired colour value the blend operator needs to know about each texture of each material as well as the material weight at every terrain surface point. Different strategies of implementing the blend operator are presented in the following sections.

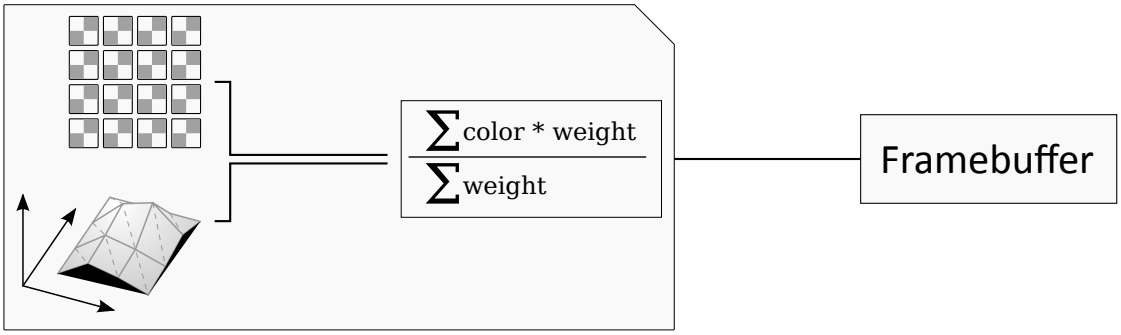
6.4.1 Blending in fragment shader

One strategy to achieve material blending would be to implement the blend operator directly in a fragment shader (Figure 14a). This approach has however a scalability issue since the shader would then have to know about every material that can influence the colour of a fragment at each surface point. Each GLSL shader has a limited number of inputs which depend on the GL implementation.

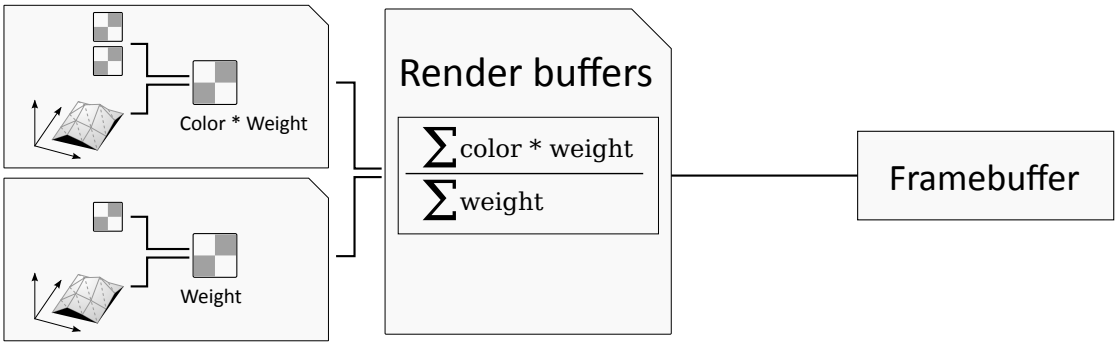
The number of inputs for the shader can be calculated as follows:

N * (4 * texture input + weight input) = 5N inputs in total, where N is the number of materials.

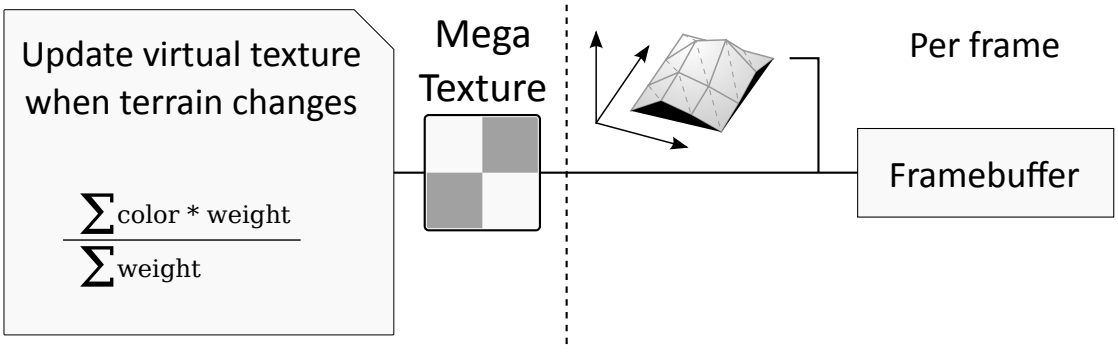
For a shader that blends four materials the number of inputs for only material blending would be 20 inputs. If no vertex attributes were used then all of these inputs would have to be provided as textures. While modern OpenGL is capable of this many texture inputs it could still potentially be a heavy operation. More materials would require proportionally heavier shader. Also, if a certain point of the terrain has fewer materials there should be optimised versions of the shader to render those areas.



(a) Direct render



(b) Pre-render to colour and accumulation texture



(c) Pre-render to a mega-texture

Figure 14. Techniques for blending materials

6.4.2 Using accumulation strategy

Another strategy to implement the blending operation could be to use an accumulation buffer for both the material colour and weight. In this strategy the terrain would be rendered multiple times to yield two buffers. In the colour accumulation buffer would be the sum of all material shader outputs weighted by the material weight at each surface point. The weight accumulation buffer would hold the sum of all material weights at the same point. These buffers could then be processed by a final output shader to perform the divide. Colour accumulation divided by the weight sum would yield the correct material blending (Figure 14b).

This deferred blending approach would be independent from the number of materials introduced by the terrain model but would most likely have a serious performance problem. The terrain geometry is often the single largest drawn element on the visible screen area. By drawing it multiple times during each frame would probably have a negative effect on the rendering frame rate. More so if the lighting calculations are performed by each material shader in the accumulation stage. This would very likely to be slower than the direct blending in the fragment shader.

6.4.3 Virtual texture approach

A third approach to implement the blend operation could be to use a large virtual texture as an intermediate buffer to blend the material inputs (15). This way the blending can be done independently from the final shading stage. The accumulation strategy is deployed to perform update operations on the virtual texture that is used at a later stage by the final fragment shader (Figure 14c).

By using this strategy the intermediate texture containing the blended terrain texture is updated only when an area of the terrain changes. The blending is not done per frame but only on demand. The final terrain texture colour can then be calculated a lot faster by using a simple fragment shader.

This approach also has its limitations. The deployed intermediate texture defines the resolution of the final terrain texture. If a single OpenGL texture is used then the maximum resolution of the intermediate texture is the maximum texture size allowed by the GL implementation. This limitation could be overcome by using a more complex strategy where more than one intermediate texture were used for areas close to the viewer and those farther away.

Another limitation is the fact that blending is done in a 2D plane. What this means

is that some rendering strategies to compensate the terrain texture stretching in high sloped areas cannot be used. One of these techniques is *triplanar texturing* in which the texture is projected along the three major axes to hide the stretching effect. (16)

6.5 The terrain geometry

The geometry of the terrain surface must be translated into primitives before it can be rendered. The rendering component of the terrain extension is responsible for doing this. There are however many ways to present the terrain geometry, some of which are more suitable to the problem than others.

6.5.1 Simple quads approach

The most straightforward implementation to draw the terrain would be to draw the terrain elevation grid using quads. In this implementation each point of the drawn quads would source their geometry from each point of the terrain model elevation data. This works for small terrain patches but is very inefficient for moderate or large scale terrains since the number of primitives needed to be drawn increase quadratically with respect to terrain size. A more optimised terrain geometry is needed to draw the terrain efficiently.

6.5.2 Optimised terrain geometry

A more efficient approach to render the terrain geometry would be the use of clip maps or a similar optimisation structure (17–19). These level-of-detail structures optimise the terrain geometry by reducing the detail level the farther away the surface is from the view point. These approaches work by storing the terrain geometry structure in graphics memory and then using a vertex shader and an elevation texture to displace the vertices. The geometry in the graphics memory is a static structure which only describes the terrain base shape and does not require manipulation per frame. Only the elevation texture needs to be updated when it changes or when the view point moves along the surface.

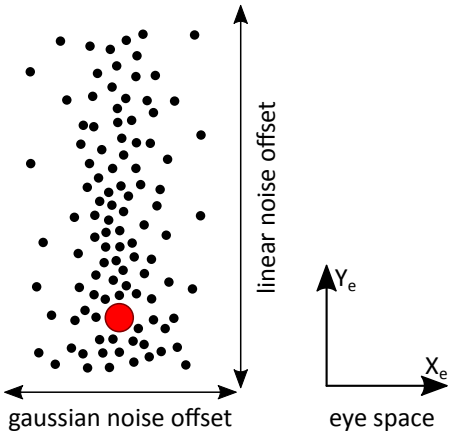


Figure 15. Direction dependent noise patterns

6.6 Rendering terrain particles

6.6.1 Particle shader

The particles of the terrain are rendered using a dedicated particle shader. There can be a significant number of particles on the move during the simulation so an efficient method of drawing them is required. The approach discussed here is a simple GLSL shader program that draws all particles using geometry instancing. The inputs to the shader program are the positions and velocities stored in a single texture. This texture needs to be updated with fresh information as the particles move. This is usually done once per frame.

The geometry instancing works by drawing one geometry multiple times. During these draw iterations the vertex shader gets an index variable that is incremented after each consecutive draw iteration. The index value is used to sample the particle property texture to get the positions and velocities for each drawn particle. With this information all particles can be drawn with just one OpenGL draw call.

6.6.2 Particle groups

The geometry instancing technique described above can be used to visualise ample number of particles but is not enough to visualise very large quantities for effects such as falling sand. The foremost problem with large quantities is the data transfer from the solver to the GPU, not necessarily the GPU performance. In essence there is a limit of how many particles can be simulated and transferred across network to the visualisation.

This situation can be improved by drawing a group of geometries around each particle. Each particle input would then be used to draw a particle group. This can be achieved by multiplying the number of geometry instances to draw by the number of particles in each group.

The particles within each group would have to be drawn with random offsets to make the result look more organic. The random offsets can be modulated with the available position and velocity data which allows more control over the visualisation of the groups.

The random data is a small problem for the shader since there usually is no real random data source available in the shader implementation. Also, the random data source must only be random with respect to each group. The random pattern must be unchanging for each individual group during all consecutive frames. Otherwise there would be unrealistic temporal movement within the geometries in each group.

To solve the random data problem a series of pre-determined random data is encoded into a texture for the shader to use. The data is not really random but for the visualisation purposes it can be perceived as such. The random data texture is sampled for each geometry within a group to get the required randomness. The sample points must be chosen carefully to get sufficient diversity to the random pattern. The draw index can be used to determine the sample point so that each group and each geometry within that group gets a different sample offset. This way the random patterns used to offset the geometries will be different for each group. If several particle groups do get the same offset pattern it can result in an unrealistic looking effect.

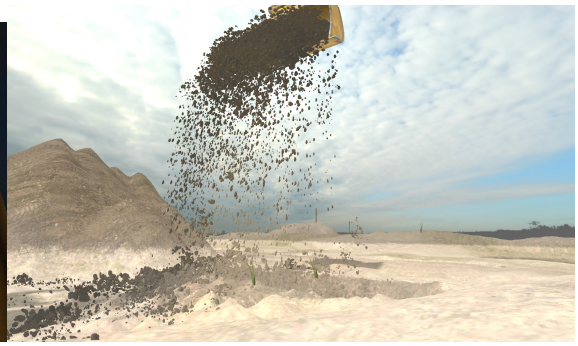
The size of the random noise texture determines the number of different offset patterns that can be achieved by using the texture. For sufficiently large textures there should however be enough variety for this not to be an issue. The noise data can be encoded into one-dimensional or a two-dimensional texture with only a couple of rows so the memory requirements are not really an issue here.

There may be a need for multiple different random data sources to achieve different offset patterns for different effects. Figure 15 shows the offset pattern using Gaussian and linear noise as random data sources.

(a) Professional excavator driver operating the simulator

(b) Manoeuvring the excavator using booms as support

(c) Operating the excavator at night



(d) Falling sand using a special shader

(e) Excavator model

Figure 16. An excavator simulator using the terrain model

7 Results

7.1 Example simulator using the terrain extension

Some pictures of a demo simulator using the terrain extension feature are presented in Figure 16 on page 39. At this point the terrain model is only exhibiting one material but it has the ability to manipulate the terrain at any point. The initial terrain data was produced with *Blender 3D software* by first modelling the required surface and then rendering it to a heightmap image. Some blur and Perlin noise filters were applied to the heightmap to yield a more organic look to the terrain. Since the terrain model is not yet able to produce subtle details on the terrain surface a diffuse texture was crafted by hand and blended on top of the terrain surface for more aesthetic look. Blender is an open source 3D creation suite (20).

The terrain can be manipulated with the slope bucket tool attached to the excavator boom. The bucket has a special 3D mesh that is used by the solver to extract soil particles from the terrain and to solve particle-object collisions. Another 3D mesh is defined for the bucket which is used by the particle shader to determine whether the particles lie inside the bucket volume.

The soil particles are rendered with a special shader that uses the technique described in Section 6.6. Figure 16d displays roughly one thousand soil particles which are duplicated with the geometry instancing technique to yield a fine effect of the falling sand. The geometry shader draws 128 visual particles for every input particle so the visual particle count is roughly around 100 000 particles.

When the soil particles are merged back into the ground their colour is gradually blended into the colour of the ground immediately below them. For this to be possible the terrain data textures are shared with the particle shader. The particle effect shader also uses this information to draw other interesting effects such as the crumbling of the ground under the excavator tracks.

The soil particle velocity affects how they are rendered which is illustrated in Figure 16d. In this case the rendered particles are stretched a small amount along the velocity direction to suggest a presence of motion blur. The particles are also scaled down proportionally to the inverse of velocity magnitude which yields a more pleasing visual appearance.

7.2 Performance

Two implementations were tested to draw the terrain geometry. The first prototype to test the terrain visualisation component was implemented using a simple 2D quad grid (see Section 6.5.1) with an ancillary geometry shader to add extra detail to the ground. The purpose of the geometry shader was to allow finer control over the perceived terrain resolution as well as to calculate surface normals using the texture height field as input. This proved to be very ineffective as expected which was seen as a dramatic drop in the running simulation frame rate. With a grid of size 512 by 512 points the running simulation was just barely above the acceptable frame rate of 60 frames per second. By adding complex graphics to the simulation the frame rate did drop below that limit. This result was foreseen however and the purpose of this prototype was mainly to confirm the functionality of the data interface and to demonstrate the correctness of the terrain simulation.

The latter implementation for drawing the terrain geometry was implemented using an approach that was comparable to the the approaches introduced in the GPU gems and Geometry Clipmaps articles (18, 19). Where the first test implementation suffered from the lack of occlusion culling and the added overhead of calculating terrain normals at every frame the new clipmaps based approach performed much better in these regards, which was expected.

Occlusion culling (or frustum culling) determines which drawable items or geometric primitives are visible from a certain view point. The purpose of occlusion culling is to identify drawable items which are not visible so that the rendering routine can skip them. Smart occlusion culling can yield considerable performance gains when a lot of unneeded geometry is excluded.

The landscape geometry that falls outside of the camera viewing frustum (its projection matrix) can be skipped. This is often more than half of the overall terrain geometry.

8 Summary

This work discussed the initial design of the terrain extension which was the main focus of a Mevea simulation software development project. The terrain model and its visualisation component presented in this work were the first part of an ongoing development project that covers many more aspects and issues than can be discussed here.

Chapter 7.1 presents an excavator simulator which demonstrates the capabilities of the terrain extension at this point of writing. Currently the terrain extension does not yet satisfy all requirements proposed in Chapter 5, but acts as a working proof of concept which will be developed further.

8.1 Learned techniques and methodologies

Considering the diverse nature of the software being developed — real-time constraints, critical code, visual requirements and large existing code base — it was challenging to design components that would satisfy all hard constraints, to design components that were extendable and compatible and finally to communicate the design ideas with the rest of the development team. This situation was improved by increasing the communication between the maintainers of the related sections of code.

A lot of time was spent in a prototyping stage. Frequently a certain design choice could not be made until a working concept of an idea was implemented and considered valid. Because of the unknown factors in the design process it was often more appropriate to use an iterative approach in the development process than to attempt to make a definitive design in the beginning.

During this project new techniques and technologies were considered for the terrain extension implementation. Some of them were specialised techniques for solving a specific problem while others fell into a more generic category of software engineering.

While many techniques were studied regarding this project not all of them were used in the released product. This was either because the technique was deemed not suitable for the problem or the technique needed a more extensive study to implement it properly.

Some notable things that were learned and used during the project can be placed into two categories:

- Algorithm and software related knowledge
- GPU debugging

The items belonging to these categories are discussed below.

8.2 Algorithm and software related knowledge

This category includes specialised approaches to efficient terrain rendering as well as basic software engineering tasks.

Efficient terrain rendering was one of the most notable techniques that had to be implemented in the terrain extension. Studying this task involved searching for papers about this subject and determining which approach was best suited to real-time simulation. Implementing one selected approach involved learning the technical details as well as how the approach had to be adapted to the existing Mevea software.

GLSL shaders were another important factor in the terrain extension implementation. Using shaders required proper knowledge of the GLSL shading language as well as knowledge of the API for managing shader inputs. More importantly, a system for the overall shader code management had to be considered since the simulation software already had a considerable number of shaders present.

Implementing the terrain and particle shaders also required supporting code structures that transferred the required data from the dynamic solver to the shaders. This included tasks such as serialisation and application internal messaging. While this supporting code is not in the scope of this document a considerable time was invested into studying the proper software development patterns to insure the resulting code was robust.

A scene graph library was used in the implementation which acted as a middle layer between the OpenGL and the application code. By using this library it was necessary to know how the scene graph organisation translated into bare OpenGL calls. Since the order and number of GL calls affect the application performance it was necessary to study both the GL API as well as the scene graph library.

Also the basics of soil dynamics simulation was studied before implementing the terrain visualisation. This was not strictly necessary but proved useful as it increased the overall understanding of the problem.

8.3 GPU debugging

Profiling and debugging tools were used in situations where the development code was performing poorly or when the result was not something that was expected. This GPU debugging required an OpenGL debugger that helped in situations

where it was necessary to inspect the contents of textures and buffers at run time. By using the OpenGL debugger it was often a lot faster to identify problems related to shader linkage and shader uniforms.

8.4 Conclusion

Developing a real-time 3D application can be a complex and involving task. Succeeding in this task does not necessarily depend only on a certain technique or algorithm but rather the overall understanding and experience with 3D systems. In a situation where for example an unusual error occurs, an experienced programmer can make a better initial guess of where to start looking for bugs.

A lot of information was accumulated during this project that cannot be listed under any specific category but which can be regarded as experience in 3D software development in general. This knowledge is not in any way less valuable than the few examples listed above.

List of Figures

1 A forwarder simulated in forest environment (Mevea Ltd.) 5
2 Extension concept 9
3 Different phenomena of ground 10
4 Simulator software components 12
5 Hardware topology 14
6 Heightmap and TIN with roughly similar memory requirements . . 21
7 Module overview 22
8 Compound data type and data separated into layers 24
9 Data storage 25
10 Data flow during simulation 27
11 Blending two textures to get a mixed texture as a result 30
12 Normalising weights 31
13 Weighted blending with and without a blend map 32
14 Techniques for blending materials 34
15 Direction dependent noise patterns 37
16 An excavator simulator using the terrain model 39

References

1. Pla-Castells, M., García, I. & Martínez, R. J. 2004. Approximation of continuous media models for granular systems using cellular automata.

2. Pla-Castells, M., García-Fernández, I. & Martínez, R. J. 2006. Interactive terrain simulation and force distribution models in sand piles.

3. What is LIDAR? 2015.
   http://oceanservice.noaa.gov/facts/lidar.html
   Accessed on 19th August 2015.

4. LandXML Home. 2015.
   http://www.landxml.org/
   Accessed on 9th March 2015.

5. Lin, P. w., Pan, Z. g., Yang, J. & Shi, J. y. 2002. Implementation of a low-cost CAVE system based on networked pc. Proc. Virtual Environment on a PC Cluster Workshop, pp. 33–40.

6. Fernando, R., Haines, E. & Sweeney, T. 2001. GPU gems: programming techniques, tips, and tricks for real-time graphics. Dimensions, 7.4, p. 816.

7. TIN in ArcGIS Pro—ArcGIS Pro | ArcGIS for Desktop. 2015.
   http://pro.arcgis.com/en/pro-app/help/data/tin/tin-in-arcgis-pro.htm
   Accessed on 17th September 2015.

8. Burbeck, S. 1992. Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc). Smalltalk-80 v2, 5.

9. Larsen, B. D. & Christensen, N. J. 2003. Real-time terrain rendering using smooth hardware optimized level of detail.

10. Bösch, J., Goswami, P. & Pajarola, R. 2009. RASTeR: Simple and efficient terrain rendering on the GPU. Eurographics Areas Papers, pp. 35–42.

11. Gobbetti, E., Marton, F., Cignoni, P., Di Benedetto, M. & Ganovelli, F. 2006. C-BDAM–Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering. Computer Graphics Forum. Vol. 25. 3. Wiley Online Library, pp. 333–342.

12. Segal, M. & Akeley, K. 2010. The OpenGL Graphics System: A Specification (Version 3.3 (Core Profile)).

13. Kessenich, J., Baldwin, D. & Rost, R. 2009. The OpenGL® Shading Language.

14. Policarpo, F., Fonseca, F. & Games, C. 2005. Deferred shading tutorial. Pontifical Catholic University of Rio de Janeiro, 31, p. 32.

15. Virtual Texture Terrain - Graphics Programming and Theory - Articles - Articles - GameDev.net. 2015.
http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/virtual-texture-terrain-r3278
Accessed on 11th May 2015.

16. GPU Gems 3 - Chapter 1. Generating Complex Procedural Terrains Using the GPU. 2015.
http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html
Accessed on 12th May 2015.

17. Livny, Y., Kogan, Z. & El-Sana, J. 2009. Seamless patches for GPU-based terrain rendering. The Visual Computer, 25.3, pp. 197–208.

18. GPU Gems - Chapter 2. Terrain Rendering Using GPU-Based Geometry Clipmaps. 2015.
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter02.html
Accessed on 8th March 2015.

19. Losasso, F. & Hoppe, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. ACM Transactions on Graphics (TOG), 23.3, pp. 769–776.

20. Home of the Blender project - Free and Open 3D Creation Software. 2015.
http://www.blender.org/
Accessed on 1st June 2015.